Synchronization/Communication techniques for OmpSs@FPGA

Master Thesis

for the award of the degree of

Master in Innovation and Research in Informatics (MIRI)

specialized in

High Performance Computing (HPC)

Miquel Vidal Piñol
(miquel.vidal@bsc.es)

Advisor
Daniel Jiménez González
(djimenez@ac.upc.edu)

Co-advisor
Xavier Martorell Bofill
(xavim@ac.upc.edu)

Computer Architecture Department (DAC)

October 24, 2017

Abstract

HPC machines are introducing more and more heterogeneity in their architecture on the road to exascale systems. The increasing complexity of the machines due to the variety of hardware architectures and accelerators makes efficient programming a task harder than ever. Heterogeneous parallel programming models, such as OmpSs@FPGA, help the programmer handle the most unfriendly parts of working with accelerators.

This master thesis analyzes the OmpSs@FPGA communication system and proposes a set of techniques to overcome the problems related to it and potentially improve the performance of the applications.

The results show that the techniques proposed speed up the applications under certain conditions and, most importantly, solves some of the limitations that had the previous communication system. In particular, the new techniques specially improve the explotation of fine-grain parallelism and open the door to explore new possibilities with regard to data communication and re-use.

Moreover, a tool (autoVivado) that automatically manages the process of bitstream generation, from the synthesis of the HLS code to the generation of the device-tree, has been developed as part of this master thesis. autoVivado has been fully integrated with the OmpSs@FPGA compiler infrastructure, providing the programmers a way to transparently generate parallel heterogenous programs and bitstreams from OmpSs applications that use FPGA accelerators.

Acknowledgments

I would like to thank my thesis advisor Dani for his valuable counsel, moral support and GREAT patience during the writing of this master thesis, and my co-advisor Xavi and Carlos for their support and helpful tips.

To my friends and colleagues at BSC. Suffering is more bearable when it's shared. Thanks for letting me disconnect from work while we argue where shall we eat or talking nonsense during lunch. Special thanks to Antonio, Jaume and Ying for their technical support.

To Jancauts, the Wacken squad, the people from Kp, my roommates and all my friends from Revistes. Thank you all.

I finalment, a la meva família. A l'Eulàlia, gràcies per estar sempre allà i aguantar-me tot aquest temps; a la meva mare Rosa i al Jaume; al meu germà Ferran i a l'Ahinoam; a la meva àvia Gina; i en especial al meu pare i al meu avi, allà on sigueu: espero que estigueu orgullosos.

This work is partially supported by the European Union H2020 program through the AXIOM project (grant ICT-01-2014 GA645496) and HiPEAC (GA687698), by the Spanish Government through Programa Severo Ochoa (SEV-2011-0067), by the Spanish Ministerio de Economía y Competitividad under contract Computación de Altas Prestaciones VII (TIN2015-65316-P), and the Departament d'Innovació, Universitats i Empresa de la Generalitat de Catalunya, under project MPEXPAR: Models de Programació i Entorns d'Execució Paral·lels (2014-SGR-1051).

Contents

List of Figures **iii**

List of Code snippets **v**

List of Tables **vi**

1 Introduction **1**
1.1 Motivation 2
1.2 Objectives 3
1.3 Contributions 3
1.4 Document organization 4

2 Background **6**
2.1 Field Programmable Gate Arrays 6
2.1.1 Xilinx Zynq-7000 SoC 7
2.2 OmpSs 9
2.2.1 OmpSs heterogeneity: the `target` construct 11
2.2.2 OmpSs@FPGA ecosystem 12

3 Related work **17**
3.1 OpenCL 17
3.2 OpenMP 18
3.3 OpenACC 19

4 Design Space Exploration **20**
4.1 Analysis of PS/PL communication interfaces 20
4.2 OmpSs@FPGA instrumentation analysis 23
4.3 Pending tasks limitation 25

5 Our Proposals: Synchronization and Communication Techniques **26**
5.1 Synchronous data transfer offloading 26
5.1.1 Synchronization and communication protocols 29

5.2 Asynchronous task management . . . 31
5.2.1 Communication protocol . . . 32
5.3 Task Batch . . . 37
5.3.1 Synchronization and communication protocols . . . 37
5.4 Asynchronous task management with task batch capability . 40

6 OmpSs@FPGA toolchain 42
6.1 Mercurium modifications . . . 43
6.1.1 Wrapper generation . . . 43
6.1.2 New Compier and Linker Flags . . . 49
6.2 autoVivado: Automatic bitstream generation . . . 50
6.2.1 Configuration file . . . 51
6.2.2 Generation step 0: HLS . . . 52
6.2.3 Generation step 1: Design . . . 53
6.2.4 Generation step 2: Synthesis . . . 55
6.2.5 Generation step 3: Implementation . . . 55
6.2.6 Generation step 4: Bitstream . . . 55
6.2.7 Generation step 5: Device tree . . . 55
6.3 Runtime modifications . . . 56
6.3.1 Asynchronous submissions of ready tasks . . . 56
6.3.2 Dynamic creation of task batch and submission . . . 57

7 Evaluation 58
7.1 Experimental setup . . . 58
7.2 Performance analysis . . . 60
7.2.1 Small-sized accelerators . . . 61
7.2.2 Medium-sized accelerators . . . 65
7.2.3 Large-sized accelerators . . . 69

8 Conclusions and Future Work 74

Acronyms 76

References 77

List of Figures

2.1 SRAM-based FPGA architecture 6
2.2 Xilinx Zynq PS/PL interfaces block diagram 8
2.3 OmpSs@FPGA compilation flow without bitstream generation 13
2.4 OmpSs@FPGA data transfers through DMA 15
2.5 OmpSs@FPGA hardware instrumentation support 16

3.1 OpenCL memory model 18

4.1 PS/PL interfaces read benchmark 22
4.2 PS/PL interfaces write benchmark 23
4.3 Original FPGA task communication model 24
4.4 Improved FPGA task communication model 24

5.1 Delayed accelerator computation due to data streaming . . . 27
5.2 OmpSs@FPGA data transfer offloading 28
5.3 Accelerator overlapping communication and computation . 29
5.4 Data transfer offload execution diagram 29
5.5 Task argument struct 30
5.6 Task information struct 31
5.7 Block design of the *Asynchronous Task Manager* 32
5.8 Asynchronous task management execution diagram 33
5.9 Asynchronous ready task struct 34
5.10 FSM of the asynchronous *Ready Task Manager* 34
5.11 *readyQueue* BRAM structure 35
5.12 Asynchronous finished task struct 36
5.13 FSM of the asynchronous *Finished Task Manager* 36
5.14 Task batch structure 37
5.15 Task batch execution diagram 38
5.16 Task batch header 38
5.17 Task header 39
5.18 FSM of the *Task Batch Manager* 40

5.19 Asynchronous task batch execution diagram 41

6.1 OmpSs@FPGA compilation flow 43
6.2 Screenshot of autoVivado help command output 51

7.1 Small-sized accelerator standalone performance 61
7.2 Small-sized accelerator standalone speedup 62
7.3 Small-sized accelerator OmpSs performance 63
7.4 Small-sized accelerator OmpSs speedup 64
7.5 Comparison of the new techniques in small-sized accelerators 64
7.6 Medium-sized accelerator standalone performance 65
7.7 Medium-sized accelerator standalone speedup 66
7.8 Medium-sized accelerator OmpSs performance 67
7.9 Medium-sized accelerator OmpSs speedup 68
7.10 Comparison of the new techniques in medium-sized accelerators 69
7.11 Large-sized accelerator standalone performance 70
7.12 Large-sized accelerator standalone speedup 70
7.13 Large-sized accelerator OmpSs performance 71
7.14 Large-sized accelerator OmpSs speedup 72
7.15 Comparison of the new techniques in large-sized accelerators 73

List of Code snippets

2.1 OmpSs application example 10
2.2 OmpSs@FPGA application example 12

6.1 OmpSs@FPGA code with Vivado HLS directives 44
6.2 OmpSs@FPGA HLS wrapper header 45
6.3 HLS wrapper reads the task info header 45
6.4 OmpSs@FPGA HLS wrapper arguments reading 46
6.5 OmpSs@FPGA HLS wrapper computation 47
6.6 OmpSs@FPGA HLS wrapper writes out dependencies . . . 47
6.7 OmpSs@FPGA HLS wrapper sends *finished* signal 48
6.8 OmpSs@FPGA HLS wrapper hardware instrumentation . . 49
6.9 autoVivado configuration file 52

List of Tables

7.1 Communication savings when exploiting small-sized accelerators cache 62
7.2 Communication savings when exploiting medium-sized accelerators cache 66
7.3 Communication savings when exploiting large-sized accelerators cache 71

Chapter 1

Introduction

As the race to reach the exascale computing continues, the use of hardware accelerators and co-processors has been steadily increasing for the past years. In the last Top500 list[1], almost 1 of every 5 systems uses an heterogeneous architecture with some kind of accelerator/co-processor, mainly Nvidia GPUs and Intel Xeon Phis; and half of the Top10 systems boost their performance using accelerators.

The utilization of hardware accelerators, such as the aforementioned GPUs and manycore processors for parallel processing, has proven beneficial both in performance and energy efficiency terms. Although the standalone performance of these technologies, as well as their interconnection with CPUs in heterogeneous architectures, has greatly improved in recent years, they seem to fall short for the ambitious project of building an exascale machine.

The next *big thing* in the accelerators community are more application-specific alternatives such as Field Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs), that have a better performance per watt than their current counterparts. ASICs provide the best energy-efficiency and application-specific performance in exchange for cost, production time and reprogrammability. On the other hand, FPGAs sacrifice part of the efficiency and performance for a more balanced trade-off between these factors.

One of the main characteristics of FPGAs, that make them an attractive alternative, is the ability to reprogram their hardware. However, this acts as

[1] June 2017 - `https://www.top500.org/lists/2017/06/`

a double-edged sword, as they require the programmer to know a Hardware Description Language (HDL), which are not widely known, and to spend a lot more time developing the logic than it would be necessary if using a high-level language.

To ease the burden of reprogrammability, High-Level Synthesis (HLS) compilers transform code written in certain high-level languages, normally C/C++, into an HDL equivalent. Additionally, some parallel programming models, such as OpenMP [1] or OmpSs [2], have added support for heterogeneous architectures in order to transparently take care of hardware-software communication and runtime execution; while other programming models, e.g. OpenCL [3] and OpenACC [4], were already designed with heterogeneity in mind.

In this work, the OmpSs ecosystem [2] is improved by analyzing and implementing new techniques for synchronization and communication between the CPU and the FPGA in the OmpSs programming model. In addition, the current OmpSs ecosystem has been fully extended with hardware generation at compile time from C code, integrating all synchronization and communication techniques proposed in this work.

1.1 Motivation

FPGAs implement their logic in arrays of Look-Up Tables (LUTs), which limit the complexity and magnitude of the designed hardware. Sometimes it is not possible to implement an entire function into hardware because it does not fit the physical limitations of the FPGA.

The usual practice to avoid this is to reduce the computational complexity and/or the data requirements of the accelerated function. Implementing a blocked version of the function algorithm reduces both factors, shrinking the size of the accelerator and leading to better and faster compilation processes. Moreover, fine-grain accelerators give the implemented hardware more flexibility to work with workloads of different sizes, and increase the application potential parallelism.

When dealing with blocked algorithms, the ideal is to maximize FPGA occupation by having multiple accelerators working concurrently, overlapping data communication and execution.

Thus, in this multi-accelerator scenario, data communication and synchronization with the CPU are two vital factors to achieve an optimal performance.

1.2 Objectives

The main objective of this master thesis is to improve the communication and synchronization between FPGA and CPU of the OmpSs programming model ecosystem for FPGA (OmpSs@FPGA). In addition to this, an important objective is to develop a fully transparent hardware generation of the bitstream from a C code, including the support for the new techniques for communicating and synchronizing. Those objectives are divided in four sub-objectives:

- Analyze the current hardware/software communication paradigm
- Propose and implement new techniques to improve communication and minimize synchronization with the CPU
- Develop an automatic tool to allow both:
  - The integration of the Mercurium source-to-source with the Xilinx toolchain, automatically generating a bitstream from C source code
  - Automatic integration of the new mechanisms of communication and synchronization with the original accelerators
- Evaluate the performance of the new techniques

By improving data communication and synchronization with the CPU, we expect to reduce overall execution times as well as obtain better overlapping between the different accelerators. Another important aspect of our proposal techniques is that those overcome the limitation of the Linux driver of accepting several Direct Memory Access (DMA) submissions.

1.3 Contributions

This master thesis has contributed to 4 conference papers and 2 journal articles:

- *The AXIOM software layers.* Microprocessors and Microsystems, 2016. [5]
- *General Purpose Task-Dependence Management Hardware for Task-Based Dataflow Programming Models.* IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2017. [6]
- *Picos, A Hardware Task-Dependence Manager for Task-Based Dataflow Programming Models.* International Conference on High Performance Computing & Simulation (HPCS), 2017. [7]
- *Implementation of the K-means algorithm on heterogeneous devices: a use case based on an industrial dataset.* ParaFPGA: Parallel Computing with FPGAs, 2017. [8]
- *The AXIOM Project: IoT on Heterogeneous Embedded Platforms.* IEEE Design & Test. Manuscript submitted for publication. [9]
- *Exploiting Parallelism on GPUs and FPGAs with OmpSs.* 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems, 2017. Manuscript in preparation. [10]

In addition, parts of this master thesis are being used on the H2020 AXIOM[2] project, on the BSC-Ikergune industrial collaboration project and on the subjects *Programació Conscient de l'Arquitectura* (PCA-GRAU) and *Supercomputing for Challenging Applications* (SCA-MIRI) of the Facultat d'Informàtica de Barcelona (FIB).

It is also planned its use on the H2020 ExaNoDe[3], EuroExa[4] and TANGO[5] projects.

1.4 Document organization

The rest of the document is organized as follows. Chapter 2 introduces the background of the main elements that conform this thesis: FPGAs

[2] `http://axiom-project.eu/`
[3] `http://exanode.eu/`
[4] `http://euroexa.eu/`
[5] `http://tango-project.eu/`

and OmpSs. Chapter 3 reviews how other parallel programming models have addressed the hardware/software communication issue. Chapter 4 presents an analysis of the OmpSs@FPGA ecosystem at a communication level. Chapter 5 describes the new techniques proposed and some implementation details. Chapter 7 presents the experimental results. Finally, Chapter 8 summarizes the main conclusions of this master thesis and presents an outlook for future work.

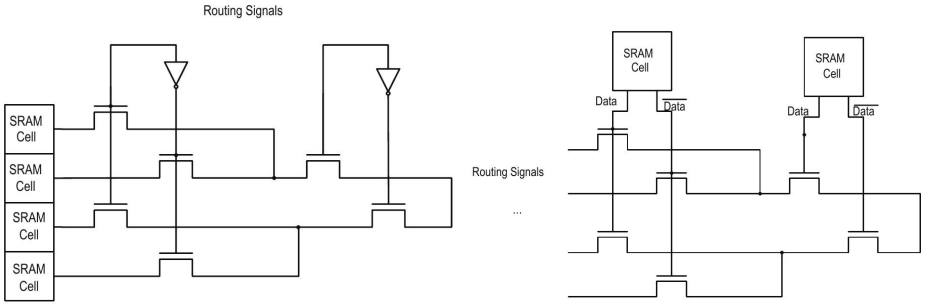
Chapter 2

Background

2.1 Field Programmable Gate Arrays

A Field Programmable Gate Array (FPGA) is a type of integrated circuit, structured as a two-dimensional array of logic blocks, whose logic can be configured after manufacturing. In each logic block, a small memory, called Look-Up Table (LUT), stores the truth table of an arbitrary n-input boolean function. The logic blocks communicate between them through a configurable interconnect network. The array is bordered by a set of I/O blocks that connect the internal FPGA logic with the I/O pins.

These three elements conform the basic architecture of an FPGA, however modern implementations also include non-programmable logic such as on-chip memory blocks, Digital Signal Processing (DSP) blocks, serial transceivers or analog-to-digital converters.



(a) SRAM cells implementing a LUT (b) SRAM interconnect multiplexer

Figure 2.1: SRAM-based FPGA architecture

Most FPGAs base their designs on rewritable memories, such as Static Random-Access Memory (SRAM) or flash, so they can be programmed more than once. The boolean function computed by each block, as well as the interconnection pattern, are defined in a *bitstream* which is loaded into the FPGA. This bitstream overwrites the values of the SRAMs, producing a different boolean function (Figure 2.1a) or interconnect path (Figure 2.1b).

In the following section, we are going to review the FPGA System-on-Chip (SoC) used in this project: the Xilinx Zynq-7000 SoC.

2.1.1 Xilinx Zynq-7000 SoC

The Zynq-7000 SoC family integrates a Processing System (PS) based on the 28nm ARMv7 32-bit Cortex-A9 and a Xilinx Programmable Logic (PL) in a single device.

To analyze and prototype the different communication/synchronization techniques we used the Xilinx Zynq-7000 ZC706 development board, featuring a dual-core ARM Cortex-A9 and a Xilinx Kintex-7 FPGA.

The main characteristics of the FPGA are the following:

- 218600 LUTs.
- 19.2Mb of Block RAM capacity in 545 36Kb blocks.
- 900 DSP slices.

The communication between the PS and the PL is done using the ARM AMBA Advanced eXtensible Interface (AXI) protocol. The board offers three communication interfaces with different characteristics: the General Purpose (GP) interface, the High-Performance (HP) interface and the Accelerator Coherency Port (ACP) interface. The internal paths of each interface can be seen in Figure 2.2.

The GP port provides non-coherent 32-bit access to the Double Data Rate (DDR) memory and the on-chip RAM. A pair of master interfaces (dark green) connected to slaves residing in the PL allow the CPU to initiate read/write transactions. Analogously, two slave interfaces (light green) are connected to masters residing in the PL, which can initiate transactions from the FPGA.

Four slave HP ports (red) provide non-coherent 64-bit access to the DDR memory and the on-chip RAM.

One slave ACP port (cyan) provides optionally-coherent 64-bit access to the DDR memory and the on-chip RAM. Cache coherency can be enabled or disabled by the programmer, writing into a control register.

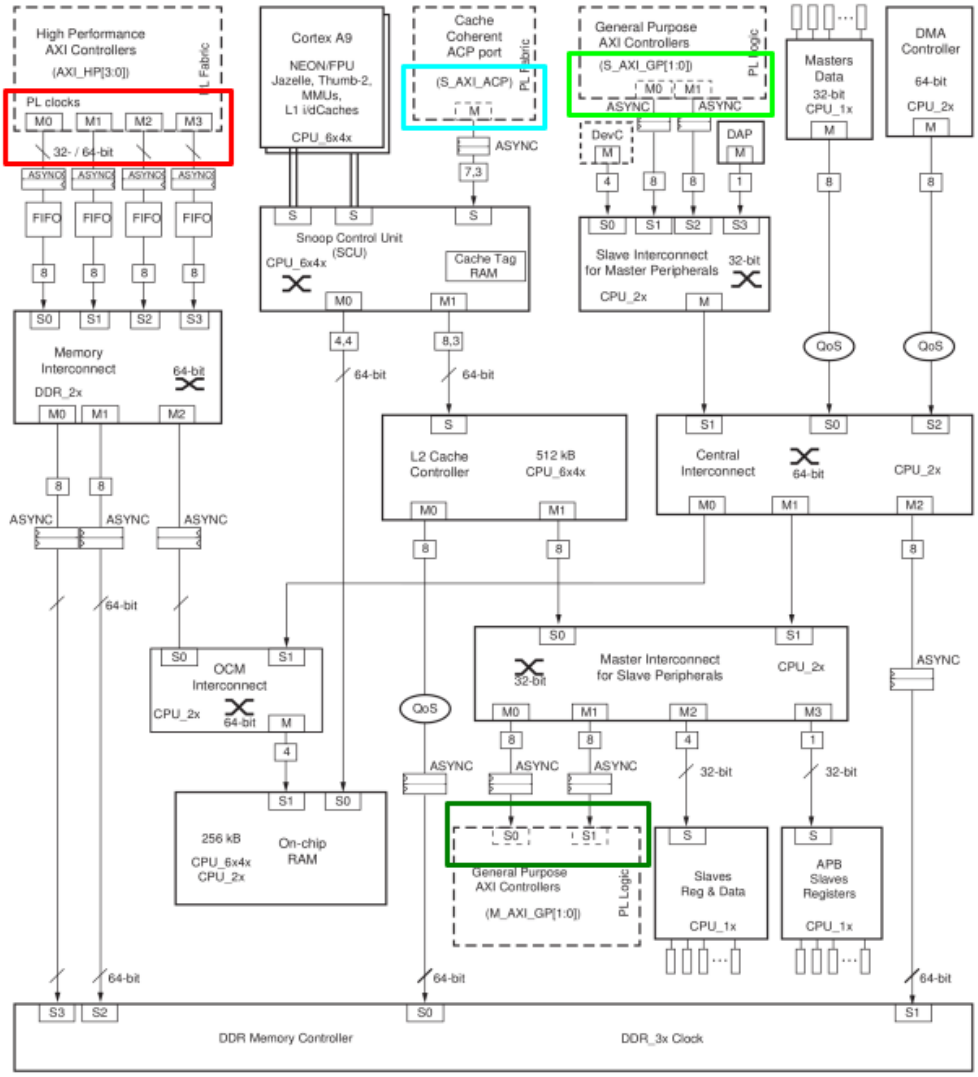


Figure 2.2: Xilinx Zynq PS/PL interfaces block diagram

The last two interfaces, HP and ACP, are more data-oriented interfaces, implementing full-duplex buses that can transfer up to 16 bytes of data at each clock cycle. GP ports are designed to send low-latency control signals, although they can be used to move data around if there is the need to.

The peak performance and suitability of the different PS/PL interfaces is described in Chapter 4.

2.2 OmpSs

The OmpSs programming model, developed by the Programming Models group at Barcelona Supercomputing Center (BSC), is an effort to integrate different features from the StarSs programming model family and push them into the OpenMP standard. It is composed of the Mercurium source-to-source compiler [11] and the Nanos++ runtime.

Its name derives from the combination of the names of the two programming models from which it takes its main design principles: OpenMP 3.0 and StarSs.

OmpSs inherits from OpenMP 3.0 the philosophy to develop parallel code: start from a sequential program and introduce certain annotations on the code, called directives, to guide the compiler on the creation of a parallel version of the code.

Diverging from the initial versions of OpenMP 3.0, which use a fork-join model, OmpSs uses a thread-pool model. It provides the `task` directive to specify regions of code that identify a unit of work, which will eventually be executed when the runtime sees fit. This approach is specially beneficial when trying to accommodate irregular applications.

Strongly influencing recent OpenMP releases, OmpSs `task` directive allows the programmer to specify data dependencies and to map the execution of certain tasks to a specific type of accelerator.

With these data dependencies, the OmpSs runtime generates a dependency graph containing all the tasks created and schedules them taking into account the given dependencies, relieving the programmer of the bur-

den of task scheduling. This significantly contrasts with OpenMP 3.0[1], where the programmer has to explicitly express how the parallel code has to be executed and synchronized between the different parts.

Code snippet 2.1 contains a simple example of a vector multiplication application that uses the OmpSs `task` directive.

```
1  #pragma omp task in(v_a[0:SIZE-1], v_b[0:SIZE-1]) out(v_c[0:SIZE-1])
2  void vector_mult(float *v_a, float *v_b, float *v_c) {
3      int i;
4      for (i = 0; i < SIZE; i++) {
5          v_c[i] = v_a[i]*v_b[i];
6      }
7  }
8
9  int main() {
10     float A[SIZE];
11     float B[SIZE];
12     float C[SIZE];
13     initialize_vector(A, SIZE);
14     initialize_vector(B, SIZE);
15     vector_mult(A, B, C);
16     #pragma omp taskwait
17 }
```

Code snippet 2.1: OmpSs application example

Each function call to *vector_mult* will correspond to a task creation with two input dependencies (*v_a* and *v_b*) and one output dependence (*v_c*), all of them of size *SIZE*. In the case of the code, function call at line 15 will correspond to a task creation with the same code of the function. This task has *A* and *B* as input data, and *C* as the output result. Line 16, with the `taskwait` directive, will wait for all task created so far, directly by the current task (parent), to finish, in this case *vector_mult* task.

In general, if there is more than one task with input and output dependencies, the runtime will take care of the correct order of the tasks based in the issue time and their dependencies.

[1]Since OpenMP 4.0, OpenMP includes tasks with input and output dependencies, strongly influenced by OmpSs works.

2.2.1 OmpSs heterogeneity: the `target` construct

Heterogeneity in OmpSs is handled through the `target` construct. It is used to specify that a given unit of work can be run in a set of devices. The construct can be applied to either a task construct or a function definition, and includes a set of clauses that allows the programmer to provide further information about the associated code.

The set of clauses currently implemented, along with their meaning, are the following:

- `device(device-name-list)` - Specifies the devices where a task can be offloaded to. If no *device-name* is specified, the default `smp` is used and it is assumed that the associated code will run on a homogeneous shared-memory multicore architecture. Current supported devices also include `opencl`, `cuda` and `fpga`.
- `copy_in(list-of-variables)` - Specifies that the shared variables in *list-of-variables* will have to be copied from the host to the device memory before the associated code can be executed. It is ignored for `smp`.
- `copy_out(list-of-variables)` - Analogous to the preceding clause, it specifies a set of shared variables that will be copied from the device memory to the host, once the associated code has finished execution.
- `copy_inout(list-of-variables)` - A combination of the previous two clauses.
- `copy_deps` - Specifies that any dependence clause attached to the associated code will have copy semantics. Any `in` will be treated as `copy_in`, `out` as `copy_out` and `inout` as `copy_inout`.
- `implements(function-name)` - Specifies that the associated code is an alternative implementation of *function-name* for the `target device`. This alternative can be used instead of the original, if the runtime considers it appropriate.
- `onto(acc_id,num_instances)` - Exclusive to `fpga` device. Specifies that the associated code will be synthesized into an FPGA accelerator identified by *acc_id*, of which *num_instances* instances will be created.

Following the example started in Code snippet 2.1, one can easily make an heterogeneous version of the application by adding a single line of code (line 1), as shown in Code snippet 2.2.

```
1  #pragma omp target device(fpga,smp) copy_deps onto(0,1)
2  #pragma omp task in(v_a[0:SIZE-1], v_b[0:SIZE-1]) out(v_c[0:SIZE-1])
3  void vector_mult(float *v_a, float *v_b, float *v_c) {
4      int i;
5      for (i = 0; i < SIZE; i++) {
6          v_c[i] = v_a[i]*v_b[i];
7      }
8  }
9
10 int main() {
11     float A[SIZE];
12     float B[SIZE];
13     float C[SIZE];
14     initialize_vector(A, SIZE);
15     initialize_vector(B, SIZE);
16     vector_mult(A, B, C);
17     #pragma omp taskwait
18 }
```

Code snippet 2.2: OmpSs@FPGA application example

Additionally, we can take advantage of the OmpSs@FPGA ecosystem to fully exploit the FPGA capabilities.

2.2.2 OmpSs@FPGA ecosystem

The development state of the OmpSs@FPGA ecosystem has changed over the course of this master thesis. Nonetheless, the initial state of the ecosystem is the one described here.

The OmpSs@FPGA ecosystem is able to generate the code that runs on the host machine with FPGA support (Figure 2.3), however there is not support to neither generate the HLS code nor the FPGA bitstream. Hardware/software communication is not handled transparently either.

Mercurium compiler analyzes the input source code and detects two main parts: the host code and the FPGA code.

The host code is transformed to include calls to the Nanos++ runtime to

spawn tasks and perform the data movements specified with the `copy_(in, out, inout)` clauses. A specially designed DMA library implements the data transfers to and from the FPGA. Then, it is compiled using the GCC compiler to run on the ARM cores of the SoC.

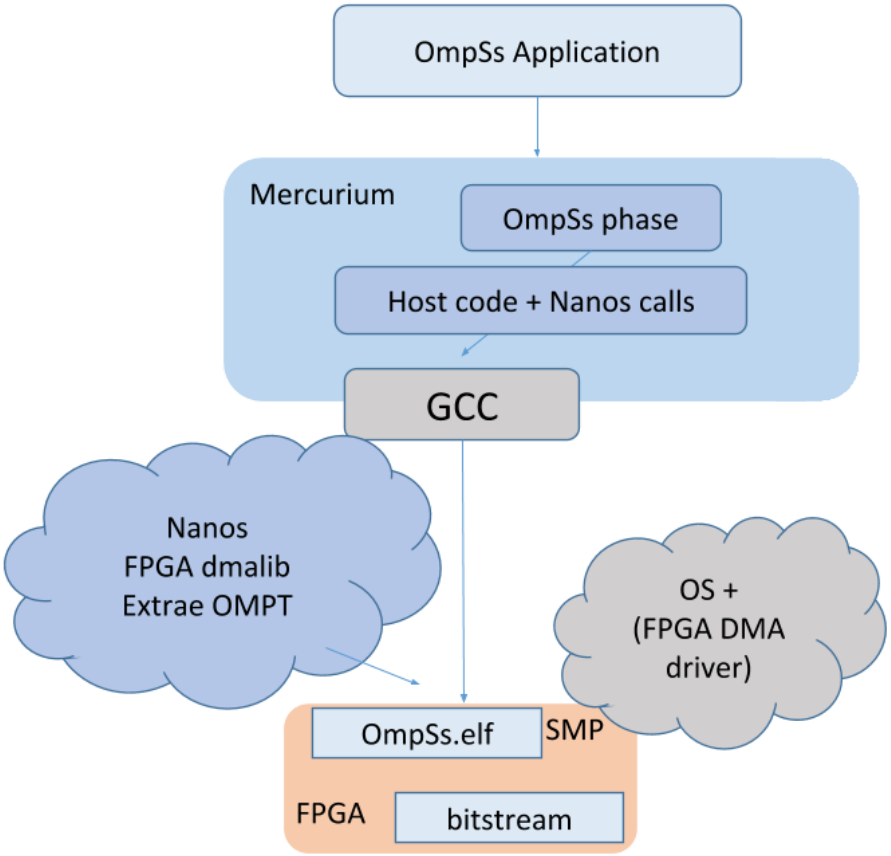


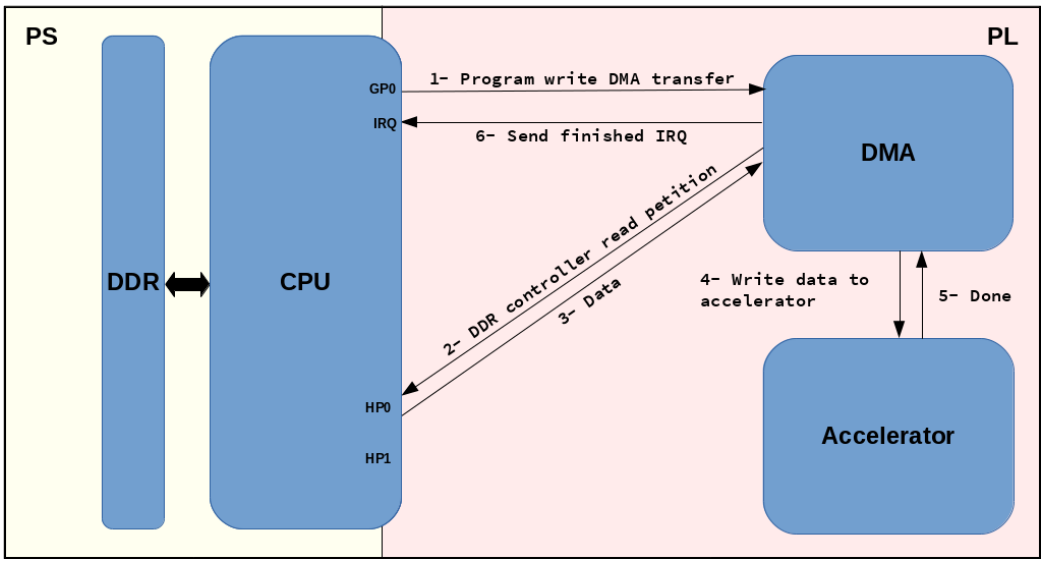
Figure 2.3: OmpSs@FPGA compilation flow without bitstream generation

On the other hand, Mercurium ignores the FPGA code, which has to be entirely written by the programmer, including all HLS directives regarding communication interfaces. Hardware design and synthesis and the generation of the bitstream through Vivado are also part of the programmers duties.

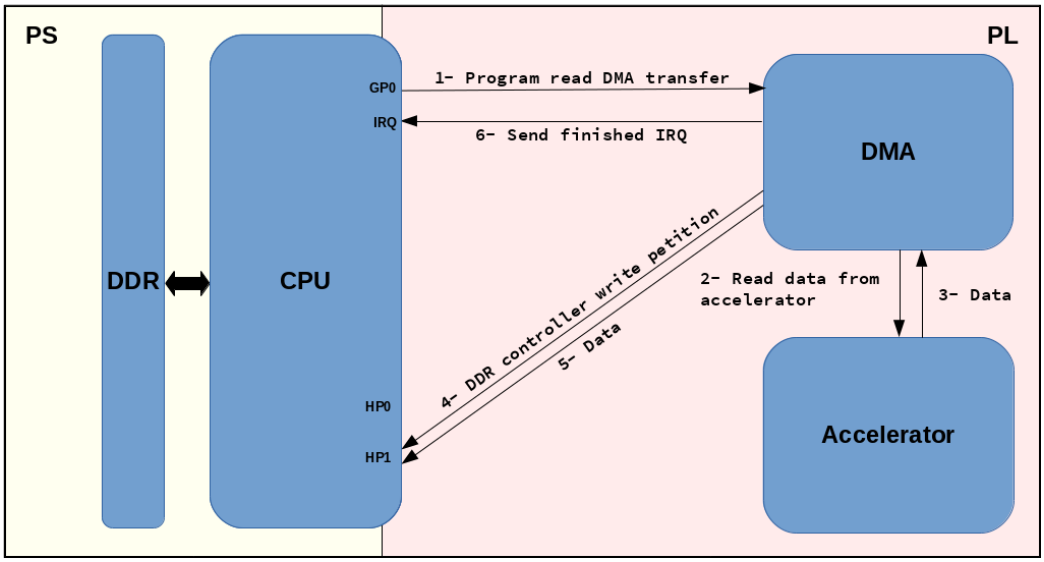
In this initial approach data communication is based on the AXI-Stream protocol. A DMA engine is necessary to transform the memory-mapped

data from the CPU to streaming data to the accelerator, and *vice versa*. The schema of the DMA transfers is shown in the diagram of Figure 2.4.

The CPU has to program the DMA transfer by writing to certain registers of the DMA engine. Data is read from the CPU memory and fed into the accelerator. Analogously, data read from the accelerator is written into the CPU memory. Both reads and writes are done in a sequential manner. Once either operation is completed, the DMA engine issues a hardware interrupt to inform the CPU.



(a) OmpSs@FPGA send data to accelerator



(b) OmpSs@FPGA retrieve data from accelerator

Figure 2.4: OmpSs@FPGA data transfers through DMA

Communication is handled on the host side by the Nanos++ runtime through calls inserted by the compiler. On the FPGA side, however, the programmer has to know how many variables and in which order have to be read or written, and insert the corresponding DMA calls in the FPGA code. This error-prone process usually implies several iterations of the compilation flow to amend those errors, increasing the already-long compilation time.

The OmpSs@FPGA ecosystem also includes instrumentation support to monitor the accelerator performance. It is based on the OMPT standard, which is a performance-monitoring interface that is being considered for integration into the OpenMP standard.

Extensions were developed for the Nanos++ runtime and the Extrae instrumentation package to comply with the OMPT standard [12].

To support the instrumentation, a hardware timer was implemented. The timer stores its current value into a BRAM memory inside the FPGA, accessible through a BRAM controller. The controller is mapped to a physical memory accessible from anywhere in the system, either from the CPU or from the accelerators.

Figure 2.5 shows a picture of a Vivado hardware design with an accelerator, the hardware timer, the CPU and their interconnection with the BRAM controller.

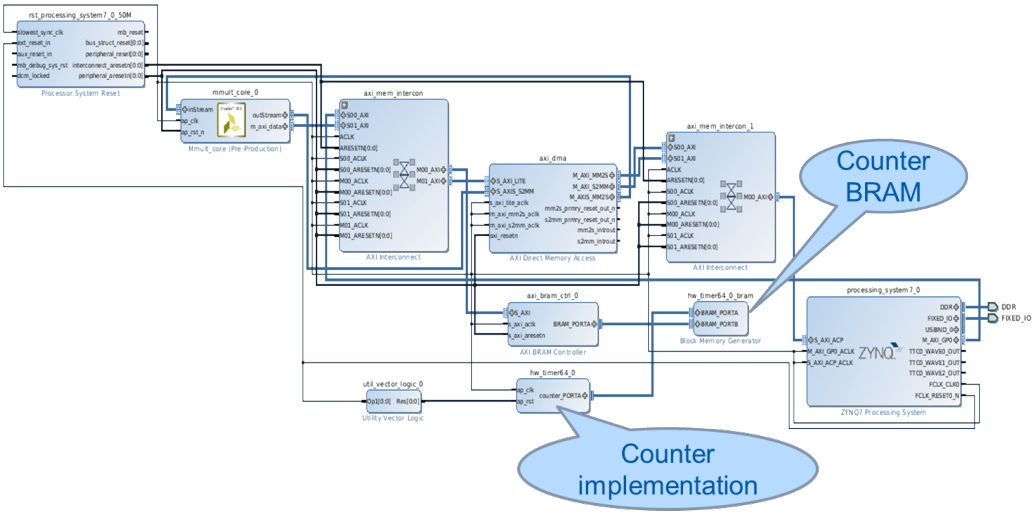


Figure 2.5: Hardware design of an IP accelerator with hardware instrumentation support

Chapter 3

Related work

In this section we are going to review how the main programming models targeting FPGA accelerators tackle the issue of communication and synchronization with the processor.

3.1 OpenCL

OpenCL implementations [13, 14, 15] mimic the memory hierarchy approach it uses for GPUs (Figure 3.1) and describe four different levels of memories: *host*, *global* + *constant*, *local* and *private*.

Host memory is any memory connected to the host processor; *global* + *constant* memory are memory chips physically connected to the FPGA which are also accessible by the processor; *local* memory is memory inside the FPGA device implemented using block RAM elements which is accessible by all accelerator Intellectual Propertys (IPs); *private* memory is memory inside the FPGA implemented using registers to minimize latency.

OpenCL FPGA kernels store their variables in *local* memory by default and the programmer has to explicitly write which variables have to be in *private* memory.

Data present on the host machine is stored in the *host* memory by default. To move data to the FPGA, the programmer must allocate space on the *global* memory, enqueue the write and read commands on the command queue and ensure that the copies finish correctly, either by issuing a blocking read/write command or waiting for the command event to finish.

The communication is done using DMA engines and is initiated by the processor. The FPGA device has a passive role in this communication model, not initiating any communication back to the CPU and writing data only to the innermost levels of the memory hierarchy.

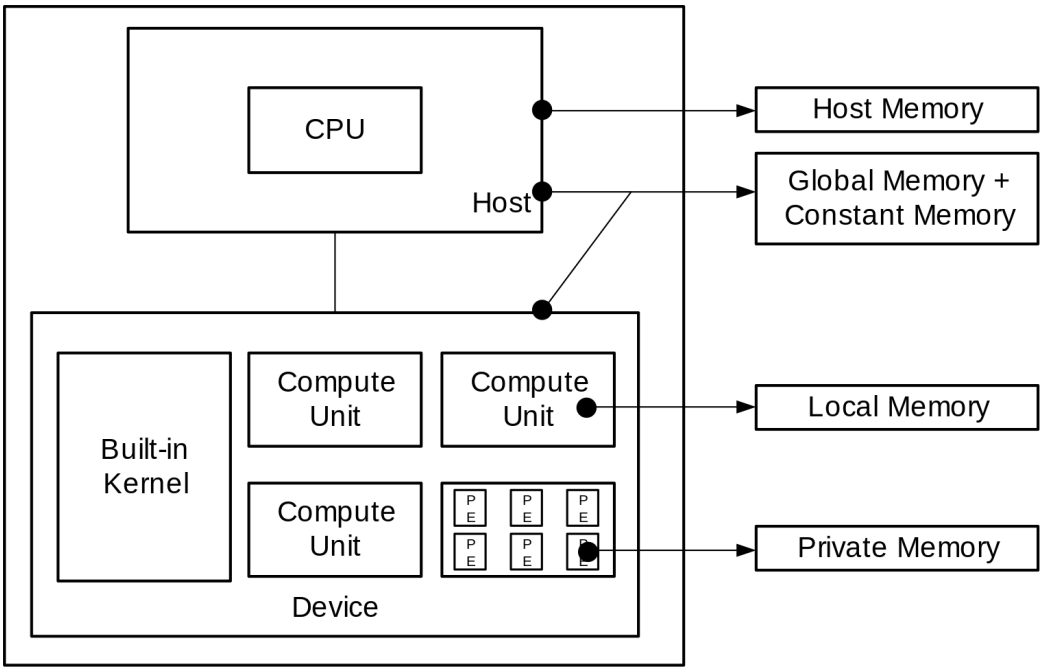


Figure 3.1: OpenCL memory model

3.2 OpenMP

OpenMP `target` directive was introduced in version 4.0 of the standard and refined in the latest 4.5 release [16]. There has been some works [17, 18] implementing the standard with similar CPU-FPGA communication approaches.

In these implementations the FPGA is understood as a passive accelerator, and all the communication is started by the host CPU. As in a GPU, local variables of accelerated functions are directly stored in the device memory; while the rest of the data is allocated in CPU memory and transferred to the FPGA through DMA engines.

In both implementations data communication is done transparently to the programmer. On [18] is handled by the Thread-PoolComposer (TPC), a

toolchain that is responsible for the synthesis of the hardware accelerators from kernel code, as well as their invocation and communication in execution time. On [17], the prototype runtime system is the one in charge of programming the DMA data transfers when the accelerator IP needs it.

3.3 OpenACC

Current OpenACC implementation for code targeting FPGA [19] relies on the source-to-source OpenARC compiler which transforms OpenACC code to an OpenCL equivalent code which is in turn compiled by the Intel OpenCL compiler [14].

Consequently, data communication follows the model already explained in Section 3.1.

Chapter 4

Design Space Exploration

In order to be able to propose a meaningful set of improvements to the communication paradigm used, a set of analysis of the aforementioned paradigm was performed. The results were obtained for a complete set of benchmarks executed both standalone (i.e. without any runtime system running) and using the OmpSs programming model.

The analysis were performed using the Xilinx Zynq-7000 ZC706 board running a Linaro Ubuntu 14.04 and a 4.6 Linux kernel. Communication between the CPU and the FPGA was done using the DMA library developed at BSC and the Xilinx DMA driver.

4.1 Analysis of PS/PL communication interfaces

We wanted to know if the choice of communication interface and protocol affects the overall communication performance. To do so we designed two benchmarking IP cores using Vivado HLS. One based on the AXI protocol and the other on the AXI-Stream protocol.

The AXI protocol is a memory mapped one, involving the concept of a target address and a certain amount of data to be transferred. It communicates directly with the DDR controller and is specially useful when working within an heterogeneous system since the FPGA has a more homogeneous view of the memory system and can manage data through memory addresses, although physical ones.

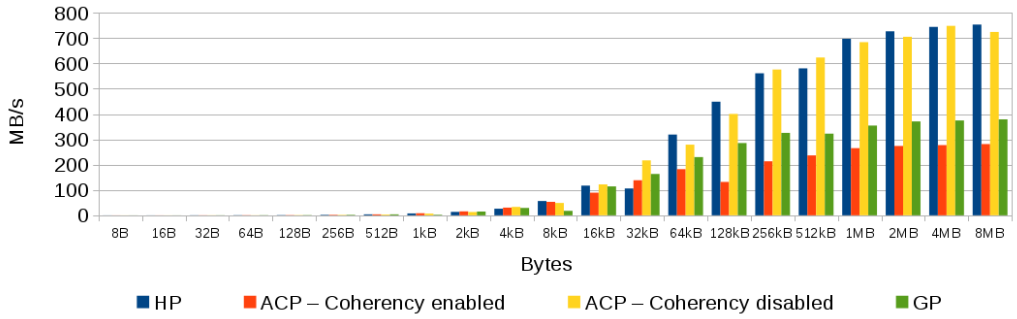
The AXI-Stream protocol acts as a single unidirectional channel where data is received in a dataflow manner. There is no concept of address

space, thus no memory addresses are present, and, when used in conjunction with a CPU, requires the utilization of DMA engines to convert memory mapped data to stream data.

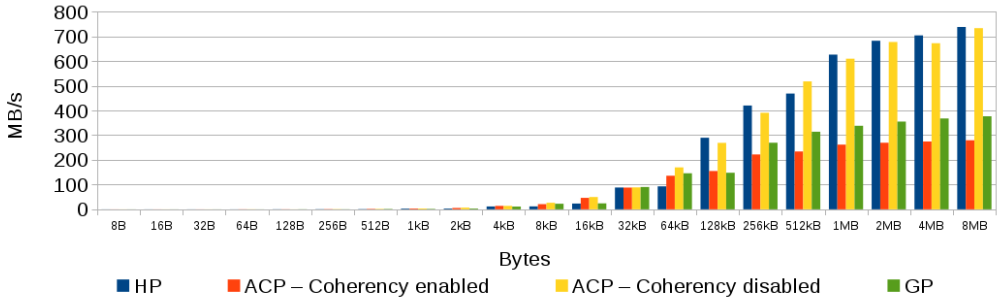
The IP cores either read (move data from PS to PL) or write (move data from PL to PS) a number of 64-bit words. The IP core based on the AXI protocol, issues petitions directly to the DDR controller inside the Zynq processor, while the one based on the AXI-Stream protocol uses an AXI DMA IP core to move data.

Several instances of the IPs were added in the bitstream and connected to a port of each of the interfaces described in Section 2.1.1. A program running in the Linux system inside the ARM cores tested the different interfaces by telling the IPs to read or write an incremental number of 64-bit words, sweeping from 2^0 to 2^{20}, i.e. from 8 bytes to 8MB, which is the maximum length that can be moved in a single DMA transfer.

AXI interfaces, whether memory mapped or streaming, are able to move a word of data each clock cycle. Word size is variable and is user-defined. On our analysis we set the word size to the maximum for each interface: 64-bit for HP and ACP and 32-bit for GP. Our FPGA was working at a clock speed of 100MHz, so the expected peak performance was approximately 763MB/s for HP and ACP and 381MB/s for GP.



(a) AXI protocol

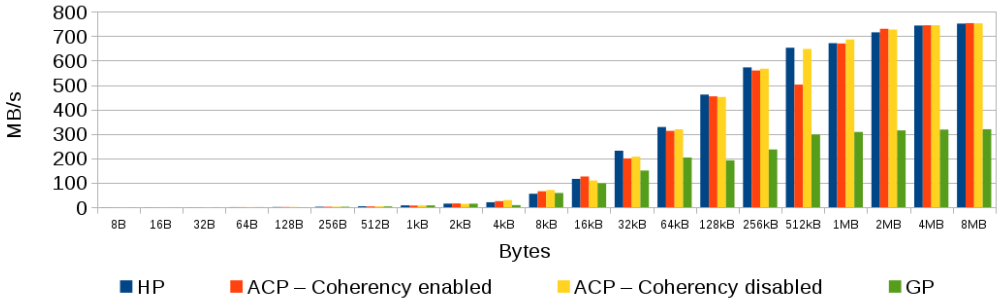


(b) AXIS protocol

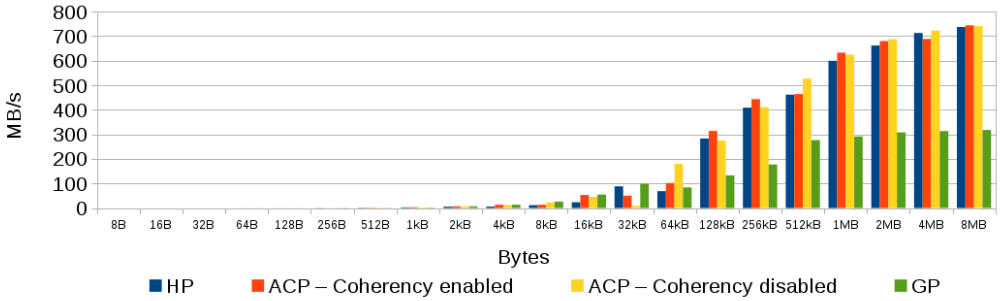
Figure 4.1: Performance of PS/PL interfaces in a read benchmark.

Figure 4.1 shows the reading performance of the PS/PL interfaces using both AXI and AXI-Stream protocols. Vertical axis shows MB/s based on the number of 64-bit words read (x-axis).

As expected, the interface with the highest bandwidth is HP (blue column), achieving near-optimal bandwidth when moving significant amounts of data; closely followed by ACP with coherency disabled (yellow column). GP (green column) also achieves near-optimal bandwidth but, since it uses a 32-bit data bus, the bandwidth achieved is half the HP one. Enabling coherency (orange column) reduces ACP bandwidth more than 50%. We could not observe any substantial difference between AXI and AXI-Stream protocols.



(a) AXI protocol



(b) AXIS protocol

Figure 4.2: Performance of PS/PL interfaces in a write benchmark.

As for the write benchmark, in Figure 4.2 we can observe that there is no substantial difference in GP, HP and non-coherent ACP compared to the read benchmark; but there is a performance boost in coherent ACP writes.

From these results we can conclude that the protocol used does not affect the performance of the interface, however the performance is greatly affected by the amount of data moved.

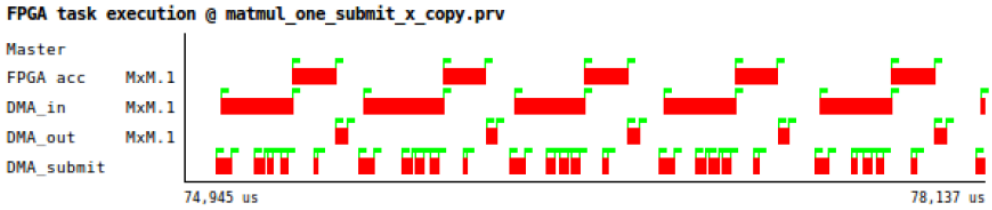
Choosing the right interface is vital when dealing with communication-intensive kernels, as it can yield huge improvements in performance. It is also important to know whether coherency is a needed feature or not, since we have seen that using a coherent interface can decrease the communication performance.

4.2 OmpSs@FPGA instrumentation analysis

Running a matrix multiply kernel instrumented with the hardware instrumentation support mentioned previously, we were able to monitor the behaviour of the accelerator.

Figure 4.3 shows an Extrae trace with several FPGA task executions of a 64x64 matrix multiply kernel. We divided the accelerator execution in three main parts, instrumenting the code accordingly: the input DMA transfers (*DMA_in*), the kernel execution (*FPGA_acc*) and the output DMA transfers *DMA_out*). *DMA_submit* corresponds to the DMA submit done by the CPU.

One different DMA submit is required per task argument copy, in or out, before the corresponding DMA transfer starts. This effectively delays the start of the kernel execution because the accelerator has to wait between each DMA submit.

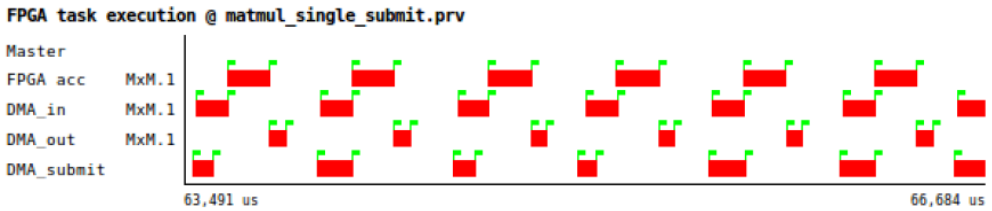


One submit per task copy (in or out) of the accelerated 64 MxM tile.

Figure 4.3: Extrae trace of an OmpSs@FPGA application submitting each task argument copy separately

In order to reduce this difference, the Nanos++ runtime was modified to provide the necessary information of the copies (in and outs) to the accelerator, in just one DMA submit. With this unique submit, the FPGA accelerator can start all the necessary DMA copies without having to wait for each DMA submit.

Figure 4.4 shows how having a single DMA submit improves FPGA communication due to shortening the waiting time in the DMA transfers. In the same time span, five full tasks could be run in the original version while, in the improved version, six full tasks could be run.



One submit per full task execution of the accelerated 64 MxM tile.

Figure 4.4: Extrae trace of an OmpSs@FPGA application submitting all task argument copies together

4.3 Pending tasks limitation

In several months of development of the OmpSs@FPGA ecosystem we run hundreds of tests and applications in several environments. We performed experiments with a multitude of implementations of both the runtime and the compiler. We also used a fair amount of different FPGA boards, Linux kernels and DMA drivers.

During all this time, we were able to observe a strange behaviour related to the number of active DMA transfers. Sometimes, specially when the number of active DMA transfers was high, some of the transfers *got lost* and did not actually take place. When this happened, it certainly meant that the results were incorrect, because some input and/or output transfers did not occur.

We suspected of a race condition and were able to narrow the problem to either the Linux kernel or the DMA driver. However we were unsuccessful on the search of the bug.

In order to avoid *losing* DMA transfers, we limited the amount of pending tasks we had at a given time, to reduce the number of active DMA transfers and the probability of triggering the bug. After several tests, we saw that the problem did not occur when having at most 4 pending FPGA tasks, and set the limit accordingly.

Chapter 5

Our Proposals: Synchronization and Communication Techniques

In this chapter we are going to review the different synchronization and communication techniques that we proposed based on the analysis previously performed.

5.1 Synchronous data transfer offloading

In Chapter 4 we analyzed the available communication protocols, AXI and AXI-Stream, and saw no difference in bandwidth between them. However, we did observe that using the AXI-Stream protocol can lead to delays on the start of the computation, due to synchronization issues with the CPU when issuing more than one input DMA transfer.

Moreover, using the AXI-Stream protocol has some implications that were not perceivable in the benchmarks. In this section, we analyze those implications and propose the best synchronization and communication mechanism between the FPGA accelerator and the CPU.

Basing the data communication in the AXI-Stream protocol implies the utilization of DMA engines to convert the memory mapped data in the CPU to streaming data. While the use of DMA engines is not bad *per se*, the data is sent by each engine in a sequential manner.

Therefore, in accelerators with several data dependencies, the accelerator would be stalled until all data has been copied to the FPGA, because the HLS compiler has no room for optimization since the data is sent sequen-

tially.

The Xilinx Vivado HLS compiler has an optimization phase where it analyzes the code to look for potential modifications in the accelerator logic to reduce the overall execution latency. For example, it can detect situations where the computation can be overlapped with data communication.

Indeed, it includes a tool to analyze the performance of the synthesized code. Figure 5.1 shows the performance analysis of a matrix multiply accelerator using AXI-Stream-based data communication. On the first row we have the control steps (not clock cycles) that guide the performance and on the second column we have the parts in which is divided the execution. Each of these parts comprises a number of logic operations, numbered in the first column, and can be understood as a set of hardware logic that implements a given function.

The reception of the matrices takes the first 6 control steps (C1-C6), two for each matrix, effectively delaying the start of the computation until control step C7. In this particular accelerator, each of the *read_matrix* functions has a latency of 1024 clock cycles, while the whole matrix multiplication takes 1156 clock cycles.

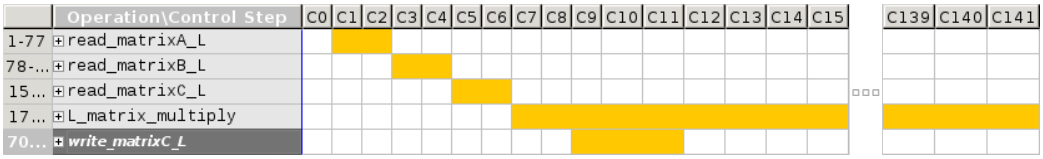


Figure 5.1: Delayed accelerator computation due to data streaming

One workaround to this problem is to instantiate a DMA engine for every variable that has to be copied to the FPGA in order for them to work concurrently and let the HLS compiler optimize the code.

This, however, has several implications:

- More FPGA resources used
- Each DMA engine uses two hardware interrupts, which are very limited (only 16 on Zynq devices)
- More overhead for the OmpSs@FPGA runtime, having to program transfers in different DMAs

- Not having a static number of DMA engines for each accelerator, and thus an homogeneous communication structure, makes them more difficult to manage

A better solution that does not suffer any of these inconveniences is the first modification that we proposed: to offload the data transfers to the FPGA and use the memory mapped AXI protocol to copy data.

A single DMA transfer, with all the data dependencies addresses, is now sent from the CPU, after which the FPGA gains control of the data communication and allows the CPU to tend to other pending tasks. Figure 5.2 shows the schema of this data transfer pattern.

We have shifted the master role in the communication process from the CPU (through the DMA engine) to the accelerator, which now issues istelf read/write petitions to the DDR controller.

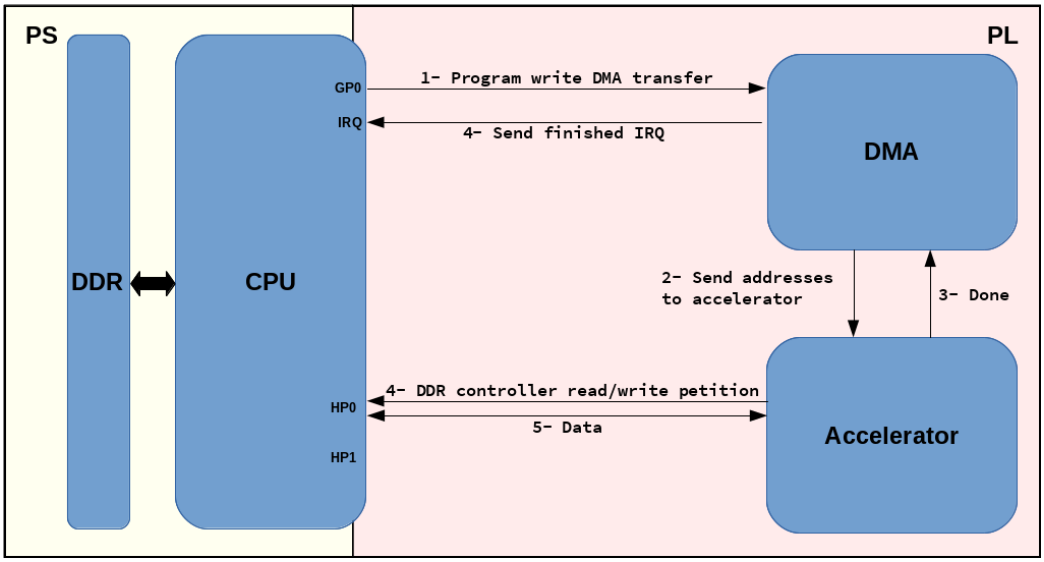


Figure 5.2: OmpSs@FPGA data transfer offloading

Additionally, we can specify a dedicated AXI port for each data dependence, since it does not imply a major increase in FPGA resource utilization. This way, the HLS compiler is able to optimize the code and overlap data communication and computation, as seen in Figure 5.3.

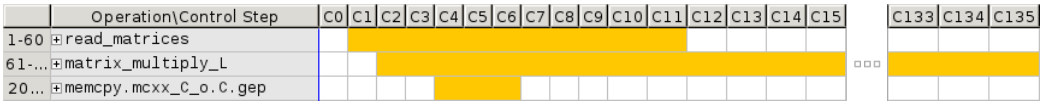


Figure 5.3: Accelerator overlapping communication and computation

5.1.1 Synchronization and communication protocols

Figure 5.4 shows the diagram of the execution of an accelerator with the data transfers offloaded.

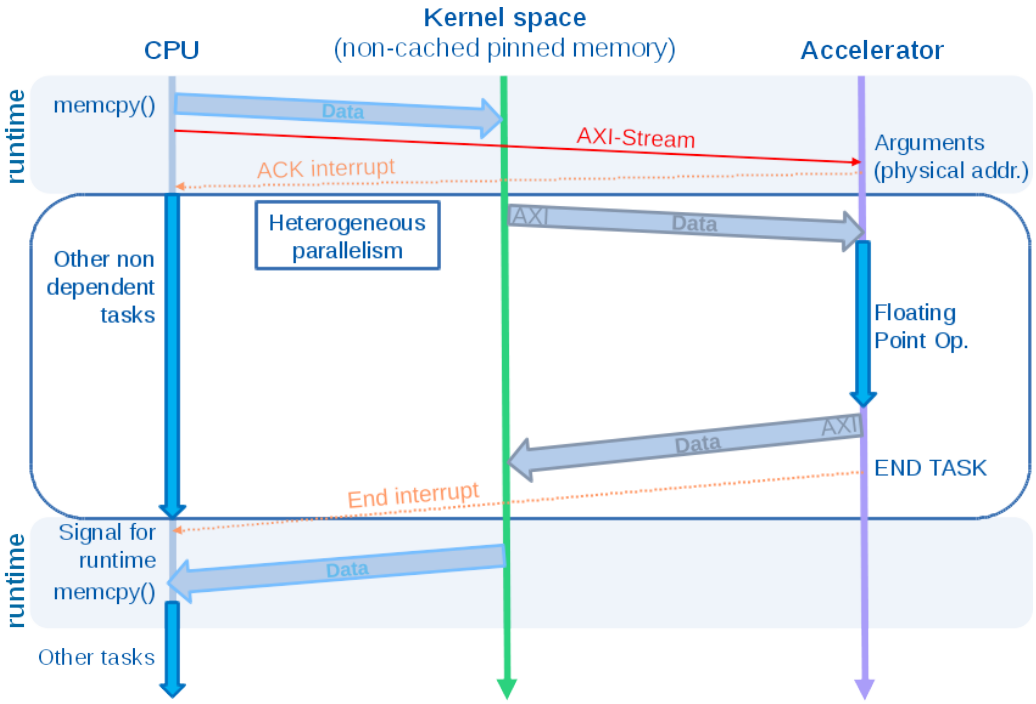


Figure 5.4: Diagram of the execution of an accelerator with data transfer offloaded

First of all, OmpSs runtime (Nanos++) copies the data dependencies that are specified with the `copy` clause to kernel space and obtains the physical address of each memory region. Virtual addresses can not be translated to physical addresses because the FPGA does not include a Memory Management Unit.

Hence the addresses that the accelerator receives have to be physical addresses. Moreover, the FPGA has no way to handle data fragmentation, so the data has to be placed contiguously in a memory region.

In addition, in order to assure that the memory pages are not physically flushed from memory, these have to be pinned. And finally, due to coherent issues of the memory used from the FPGA, shared with the CPU cores, the memory pages have to be non-cached memory.

Inside the accelerator, arguments are identified by a number, starting from 0. The addresses are sent paired with the number that identifies the argument they represent, so they can be read in any given order.

Once Nanos++ have obtained the physical addresses, it stores them into a 128-bit struct, along with the argument identifier (*argID*) and *argCached*, which is a flag that specifies if the argument is cached inside the accelerator and its copy can be avoided.

Figure 5.5 shows the memory organization of each of those fields of an argument. This struct is organized in two 64-bit words. The 64 lowest most significant bits contains the *argID* and the *argCached* fields, and the 64 most significant bits stores the argument address (*argAddr*).

Figure 5.5: Task argument struct

Each argument struct is then stored in the task information struct that represents the entire task. The struct is conformed by the task header and the task arguments. Figure 5.6 shows the bit organization of this structure in 64-bit words.

The task header contains information that is not part of the accelerated function, but is used by the accelerator to modify its behaviour. It includes the task identifier *taskID*; the addresses to the hardware counter and the instrumentation buffer *instrCounterAddr* and *instrBufferAddr*, only present if the hardware support for instrumentation is enabled; the destination identifier *destID*, used to guide interconnection inside the FPGA; and the *compute* flag, used to activate or deactivate the computation part of the accelerator.

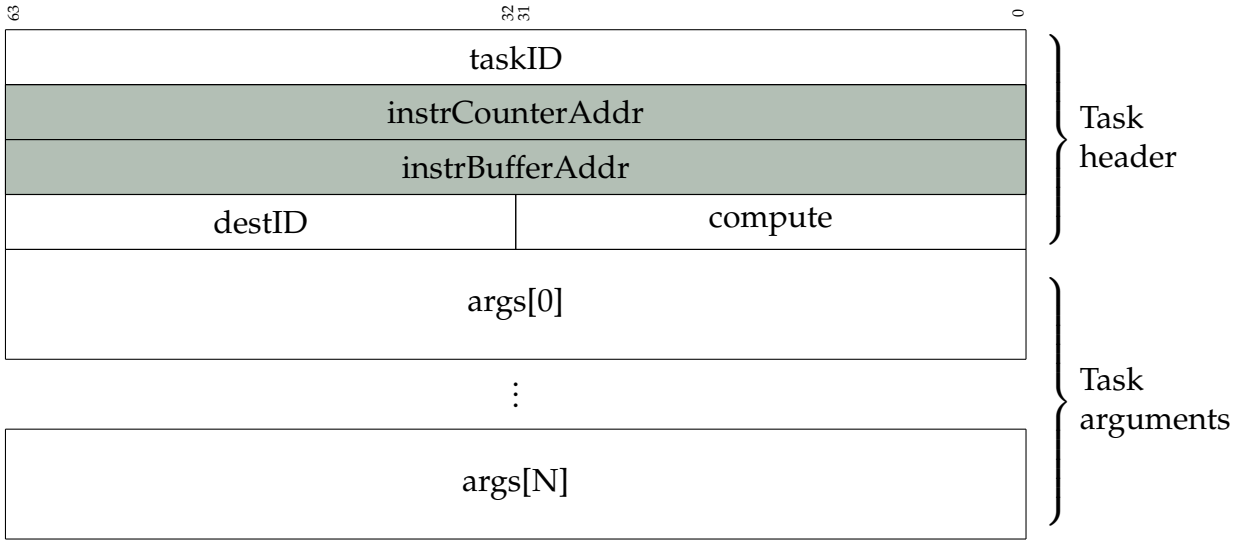


Figure 5.6: Task information struct

Finally, the runtime sends the struct through the DMA in a single transfer and waits for its completion.

Once the accelerator has received the task info struct, the CPU is informed through the DMA hardware interrupt that the transfer has finished and can continue working. The accelerator, at the same time, issues a read petition to the DDR controller and receives the input data dependencies.

Then, the accelerator performs the task computation and writes the output dependencies back to kernel space. To finish its role in the task execution, the accelerator sends through DMA the identifier of the task that has executed, *taskID*, to inform the CPU of the completion of the task.

Nanos++ receives this signal and copies the output data dependencies that are specified with a `copy` clause from kernel space to user space.

5.2 Asynchronous task management

In this section we present a mechanism to completely remove the explicit synchronization between CPU and FPGA. This mechanism gets rid of the remaining DMA engines in our communication system and overcome any OS limitation in the number of DMA transfers in flight. It is built on top of the technique described in the previous section, and it uses the same

memory structs.

From the point of view of the Nanos++ runtime, each accelerator was represented as an instance of a DMA engine where to send the task information (whether whole data or memory addresses). We decoupled the set of accelerators from the CPU by removing the DMAs and adding a specific hardware to manage ready tasks, submit them into their corresponding accelerator and handle the finished ones.

Figure 5.7 shows the block design of the manager, called *Asynchronous Task Manager*, which is conformed by 5 sub-components: the *Ready Task Manager*, the *Finished Task Manager*, the *readyQueue* BRAM, the *finishedQueue* BRAM and the *accAvailability* BRAM.

The *Ready Task Manager* has three AXI connections: to the *readyQueue* BRAM, to the *accAvailability* BRAM and to the SoC DDR memory, and an AXI-Stream output that connects with the accelerators through a crossbar.

On the other hand, the *Finished Task Manager* has two AXI connections: to the *finishedQueue* BRAM and to the *accAvailability* BRAM, and an input AXI-Stream port connected to the accelerators through a crossbar.

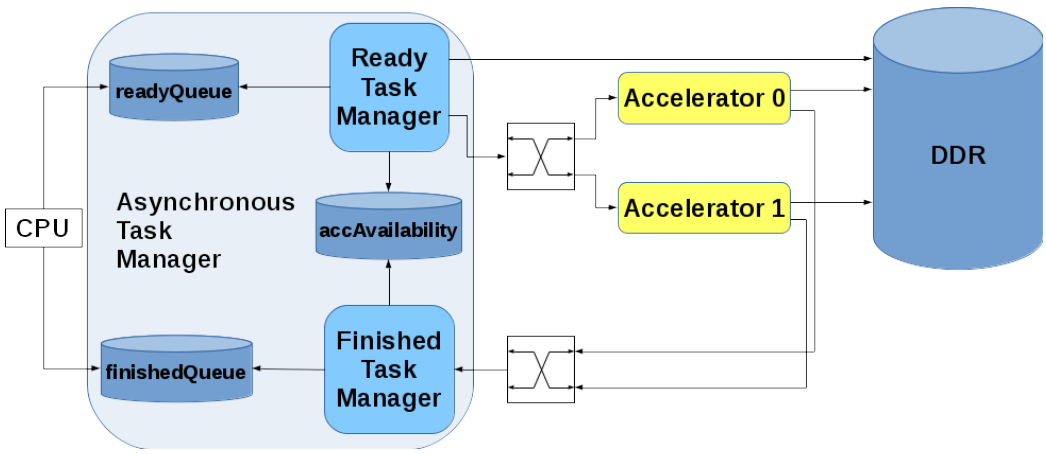


Figure 5.7: Block design of the *Asynchronous Task Manager*

5.2.1 Communication protocol

Figure 5.8 shows the diagram of the execution of an accelerator asynchronous task management. Observe that no explicit synchronization be-

tween the CPU and the FPGA remains. All interaction is done through asynchronous accesses to kernel space memory.

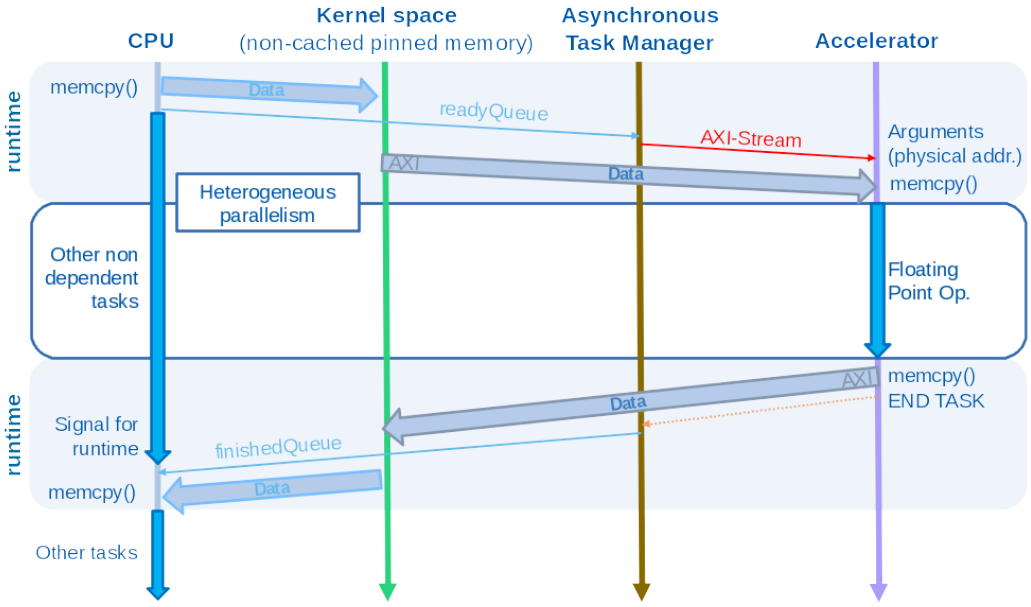


Figure 5.8: Diagram of the execution of an accelerator with asynchronous task management

Nanos++ runtime follows the same procedure as the one explained in the previous section: it copies all the data dependencies specified with a `copy` clause to kernel space, packs the physical address of each data dependence in the task argument struct (Figure 5.5) and then stores each argument in the task information struct (Figure 5.6).

Once the task information struct has been generated, the runtime obtains its physical address and writes it into the asynchronous ready task struct and stores it in a free slot of the *readyQueue* BRAM.

Figure 5.9 shows the 64-bit word organization of the asynchronous ready task struct. The less significant word stores the task information struct physical address (*taskInfoAddr*), and the most significant word stores a bit-mask, *argsBitmask*, that marks which arguments are ready and can be sent to the accelerator; *size*, that specifies the number of 64-bit words that will be sent to the accelerator (task header + task arguments); the identifier of the accelerator where to send the task, *accID*; and a *valid* field to indicate if the struct represents a valid task or not.

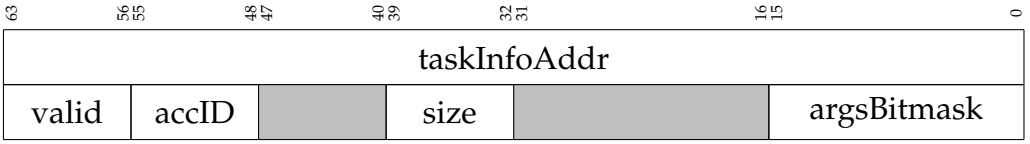


Figure 5.9: Asynchronous ready task struct

At the moment, the *argsBitmask* field is not used and we consider that all the arguments are ready. However it could be used in future versions of the manager to prematurely send some arguments to accelerators in a sort of prefetching.

At the same time the runtime is initializing the task structs, inside the FPGA, the *Ready Task Manager* is already waiting for ready tasks. Its behaviour is synthesized in the FSM of Figure 5.10.

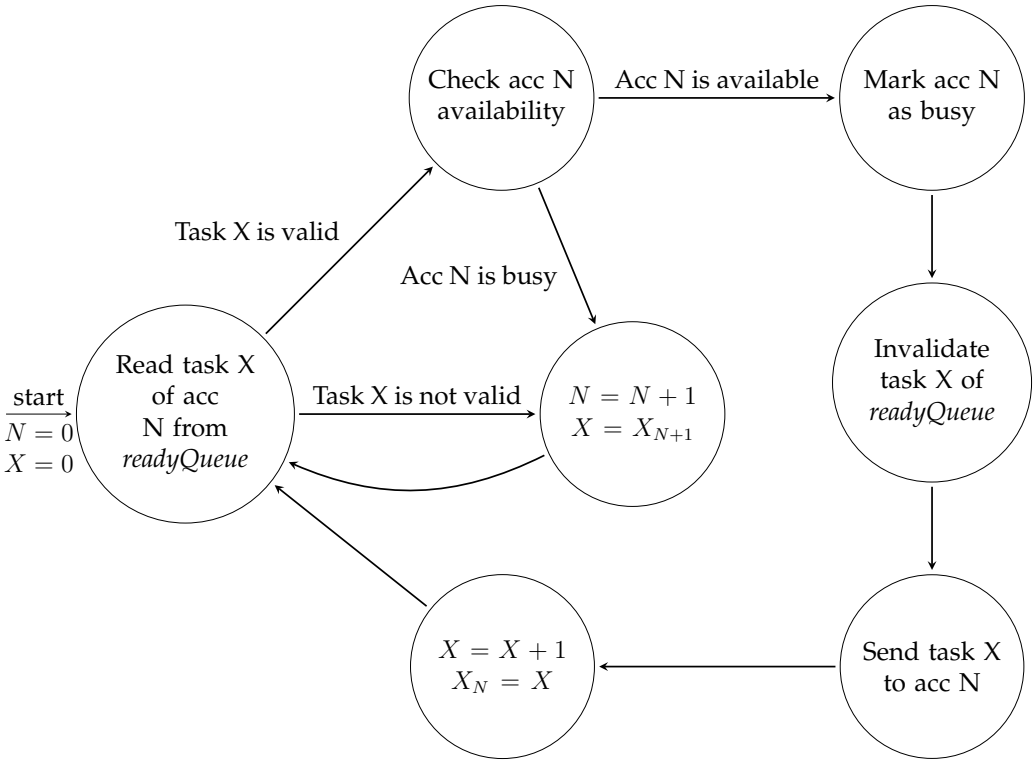


Figure 5.10: Finite State Machine of the asynchronous *Ready Task Manager*

The manager implements an active polling over the *readyQueue* BRAM, looking for valid ready tasks. The BRAM is divided in a number of regions and each of those represents a different accelerator. Figure 5.11 shows

the current BRAM structure, containing 1024 slots divided in 16 regions. Therefore, each accelerator has room for 64 pending tasks. The size of the BRAM and the number of regions can be tuned in the source code of the manager. The search for valid tasks is done following a Round-robin algorithm through every region.

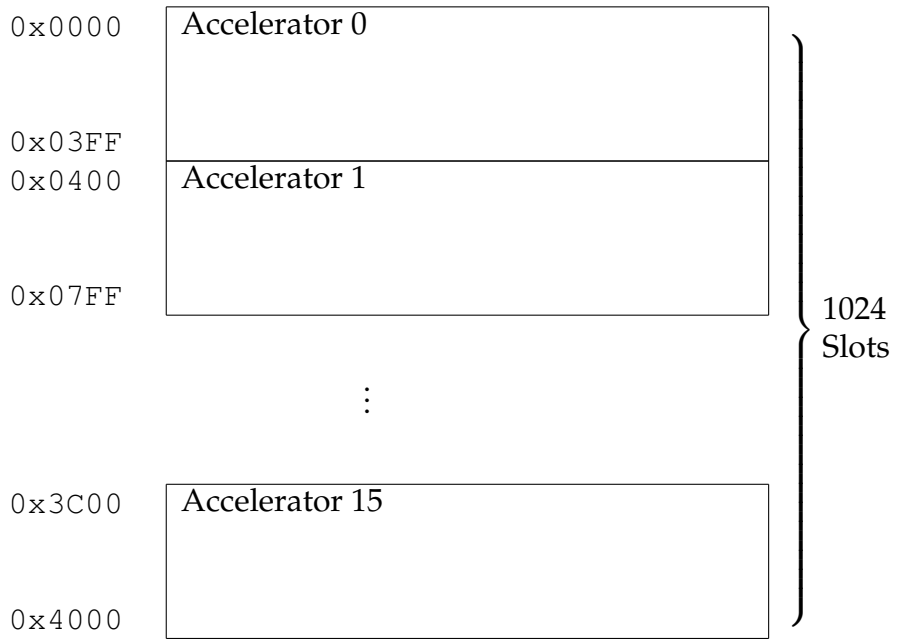


Figure 5.11: Division of the *readyQueue* BRAM in regions.

When the manager finds a valid task, it checks whether the accelerator that has to execute it is available. The accelerators availability status is stored in the *accAvailability* BRAM, which the manager checks out.

If the accelerator is busy, the manager continues the search for valid tasks for the next accelerator in the Round-robin schedule. If it is available, the manager marks the accelerator as busy in the *accAvailability* BRAM, sets to 0 the *valid* field of the ready task struct in the *readyQueue* BRAM and sends the task to the given accelerator.

Concurrently, *Finished Task Manager* waits for the reception of a *finished* signal from one of the accelerators, indicating the completion of a task. Then, it marks the accelerator as available in the *accAvailability* BRAM, searches an empty slot in *finishedQueue* and stores the finished task information in

the asynchronous finished task struct (Figure 5.12).

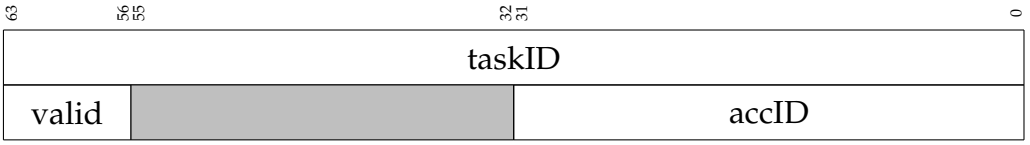


Figure 5.12: Asynchronous finished task struct

The struct contains the task identifier *taskID* and the accelerator identifier that has executed it, *accID*.

Finished Task Manager behaviour is synthesized in the FSM of Figure 5.13.

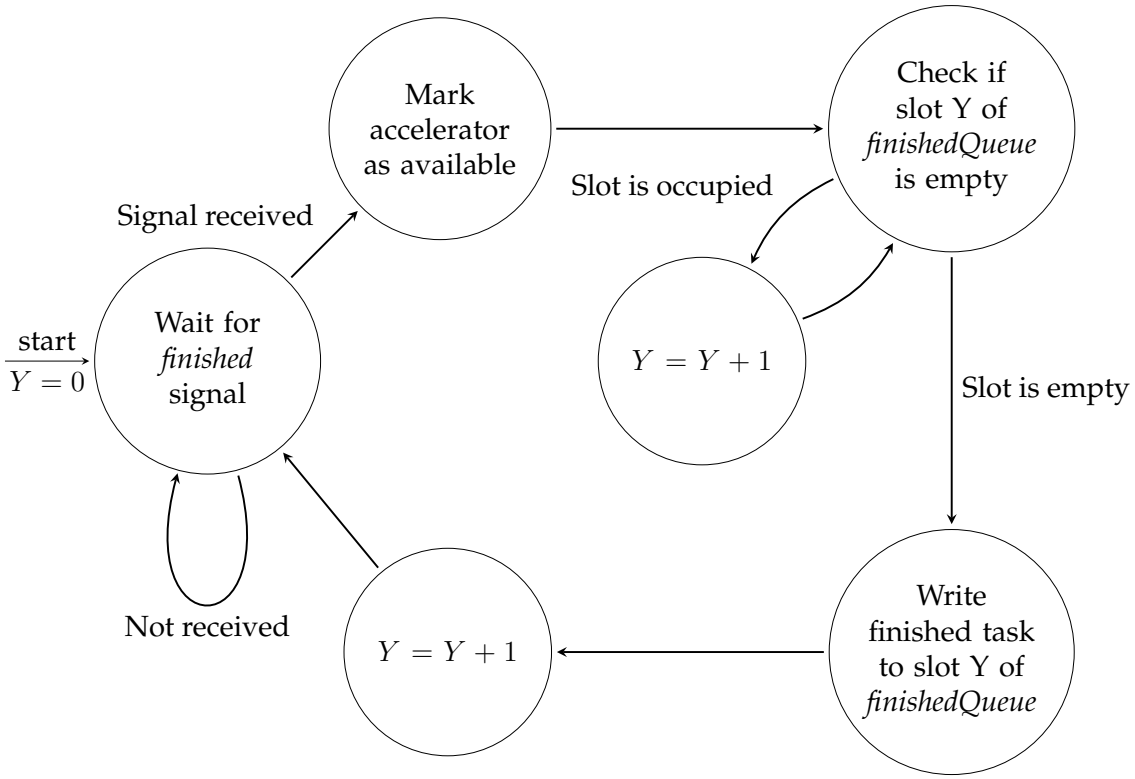


Figure 5.13: Finite State Machine of the asynchronous *Finished Task Manager*

5.3 Task Batch

In parallel with the *Asynchronous Task Manager* modification, we proposed another extension to the communication structure described in Section 5.1.

With the objective of reducing DMA transfers and CPU/FPGA synchronization, we developed the concept of FPGA task batching.

A task batch is a special task that does not contain code inside, but a number of smaller tasks. Its purpose is to be fed into a hardware manager that reads the task batch information and proceeds to submit the inner tasks to the accelerators. Figure 5.14 shows our definition of task batch: a set of interrelated tasks with data dependencies between consecutive tasks.

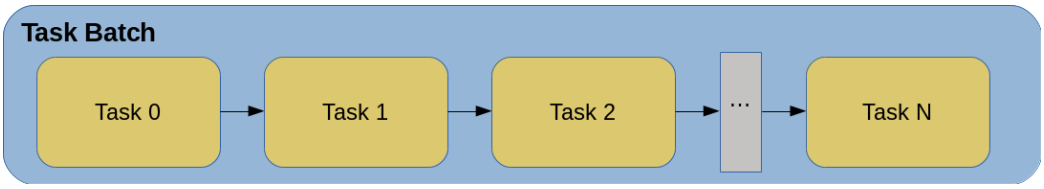


Figure 5.14: Structure of a task batch

Moreover, defining this type of structure has two additional benefits. Since we know that the set of tasks must be executed one after the other, the Nanos++ runtime system does not need to insert the tasks into the dependency graph, reducing the overall OmpSs overhead. In addition, if we know which data is shared among the tasks, we can re-use it by setting the *argCached* flag and avoid unnecessary data copies.

For example, on a blocked matrix multiply algorithm, we would have a 3-level nested loop iterating over the different blocks. The innermost loop fits exactly the structure of our task batch definition, so we could group all the iterations of the loop in a single task batch.

All those tasks share one of the three matrix blocks that is read by the first task and used by the following ones. We can re-use that block by marking it as cached in the *argCached* field in the task argument struct (Figure 5.5).

5.3.1 Synchronization and communication protocols

The runtime system stores the data dependencies in kernel space the same way as in Section 5.1, but instead of storing the data of just one task, it

stores the data dependencies of the whole set of tasks that pertain to the task batch.

Batching tasks relieves the CPU of all the management related to the inner tasks (i.e. handling dependencies, task scheduling, data copies..) and allows it to focus on other non-dependent tasks (Figure 5.15).

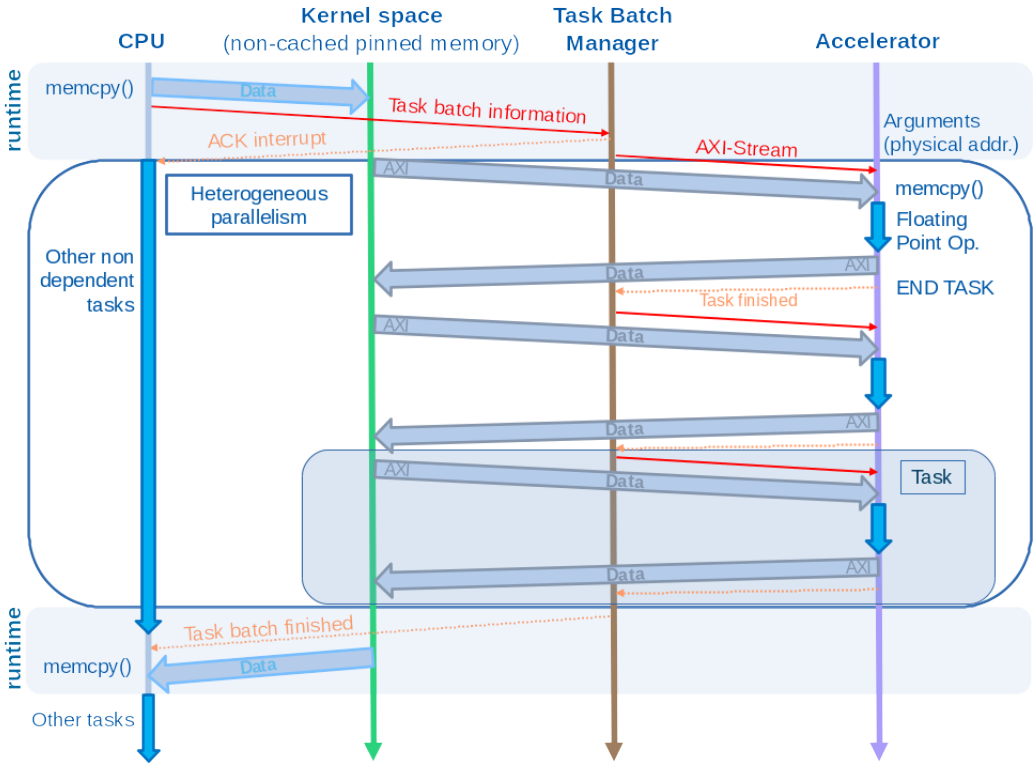


Figure 5.15: Diagram of the execution of a task batch

The hardware manager, called *Task Batch Manager*, receives the task batch header (Figure 5.16) that contains the task batch identifier *taskBatchID*; the number of tasks within the batch, *numTasks*; and the destination identifier, *destID*, which drives intra-FPGA communication.

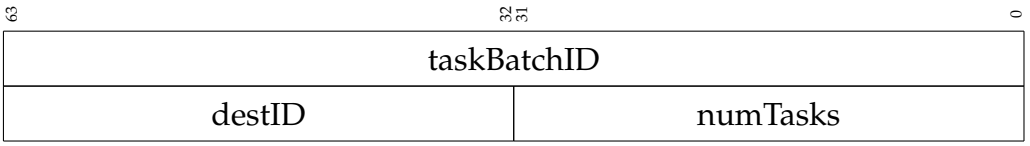


Figure 5.16: Task batch header

Each task within the batch is represented as a task header (Figure 5.17) and a task information struct (Figure 5.6). The task header contains a bitmask that marks which arguments are ready, *argsBitmask*; the number of arguments of the task, *size*; and the accelerator that has to execute the task, *accID*

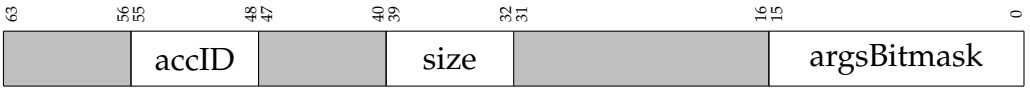


Figure 5.17: Task header

The manager follows the FSM of Figure 5.18. It first reads the header of the first task. Next it reads each element of the task information struct and checks *argsBitmask* to see if the argument is ready. If it is ready, the manager sends it to the accelerator. Analogously, if the argument is not ready, the manager stores the argument in a temporary buffer in order to send it afterwards.

In contrast with the previous section, where the *argsBitmask* field is not used, here we can actually use it to send arguments that are ready from a task that can not be executed yet, to idle accelerators.

The first task of every task batch is always a ready task, meaning that all its input dependencies are met, so the manager always sends its arguments right away.

Once the first task has been sent, the manager starts reading the remaining tasks of the batch, sending the arguments that are ready and storing in temporary buffers the ones that are not.

After all the tasks have been processed, the manager waits for the *finished* signal of the first task of the batch and proceeds to send the arguments of the second one that are stored in the temporary buffer. It then waits for this second task to finish, and repeats the process until all the tasks have been send to the corresponding accelerators.

Finally, when the *finished* signal of the last task arrives, the manager sends its own *finished* signal to the CPU to inform that the task batch has been executed in its entirety and can read the output data from kernel space.

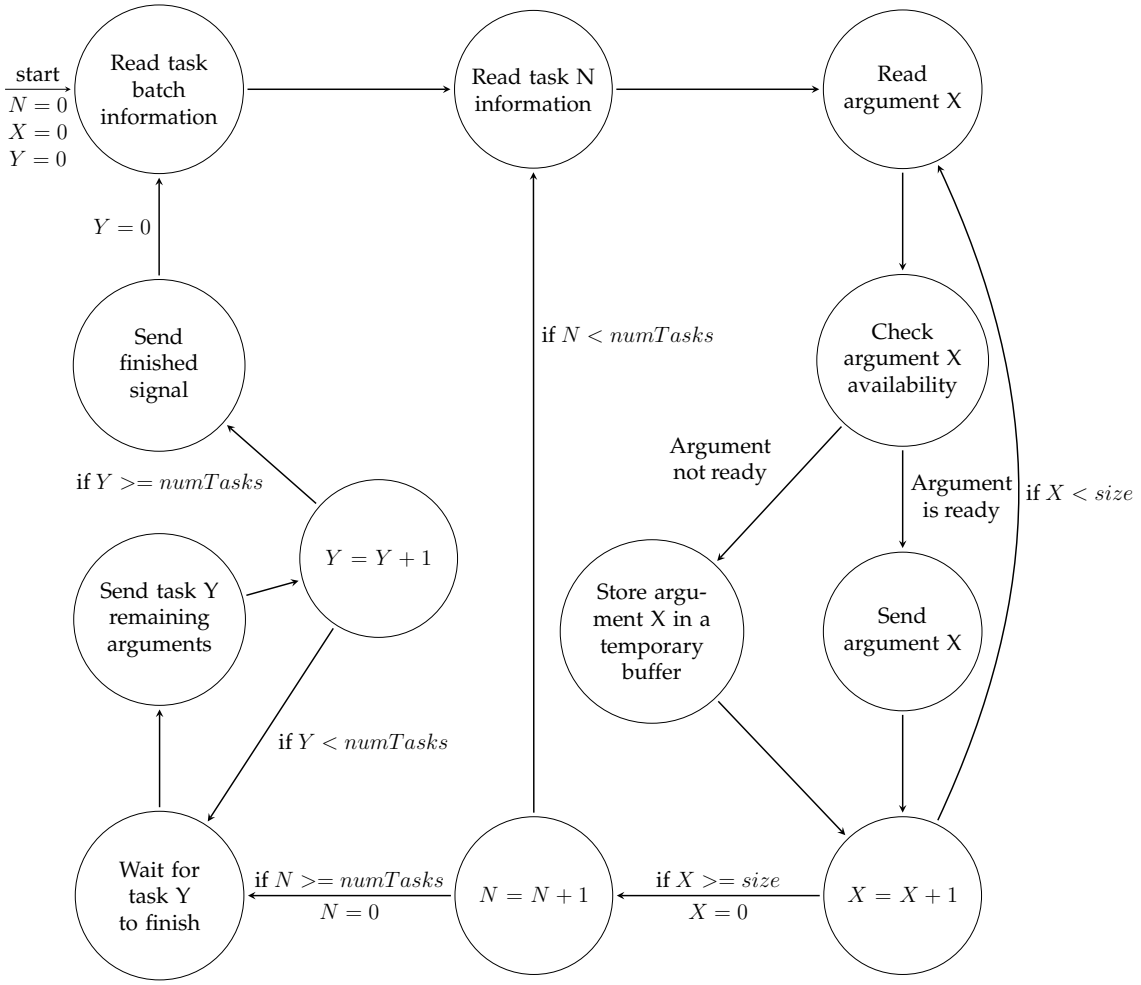


Figure 5.18: Finite State Machine of the *Task Batch Manager*

5.4 Asynchronous task management with task batch capability

Finally, we proposed the combination of the three modifications described: an asynchronous task batch with offloaded data transfers.

The tasks are packed in a task batch struct, the same way as it is explained in Section 5.3, then this struct is stored in kernel space and its physical address written into a slot of the *readyQueue* in order to feed the *Asynchronous Task Manager*.

The *Asynchronous Task Manager* will read the task batch from kernel space and feed, in turn, the *Task Batch Manager*, which will, ultimately, send the tasks to the accelerators (Figure 5.19).

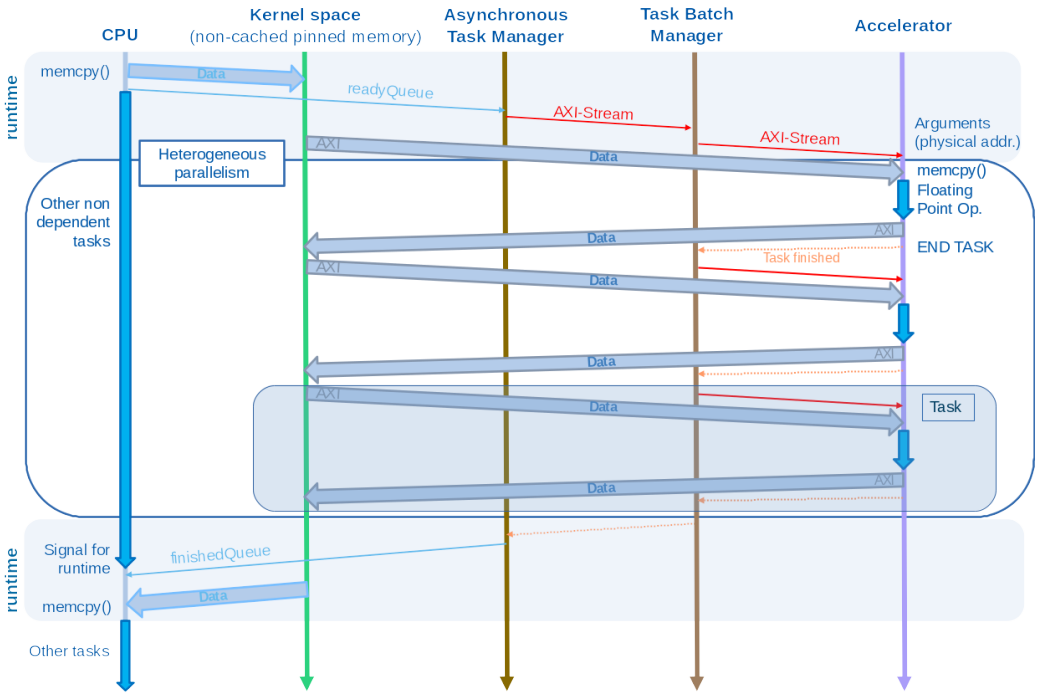


Figure 5.19: Diagram of the execution of an asynchronous task batch

Chapter 6

OmpSs@FPGA toolchain

To accommodate the work of this master thesis, we developed the whole OmpSs@FPGA ecosystem [2] to fully support the automatic bitstream generation and transparent handling of CPU/FPGA communication.

Prior to this work, OmpSs@FPGA implementation only supported 32-bit Xilinx Zynq SoC families and required the programmer to code the FPGA code by hand and manually use the Xilinx toolchain (Vivado HLS + Vivado) to generate the FPGA bitstream.

Figure 6.1 shows the full OmpSs@FPGA ecosystem with support for automatic HLS code and bitstream generation. The left branch is what was already developed of the OmpSs@FPGA ecosystem at the start of the master thesis and the right branch is what has been developed in the context of this work.

The left branch represents the part of Mercurium that generates the host code of the OmpSs@FPGA application, inserting calls to the Nanos++ runtime system and the DMA library.

The right branch represents the Mercurium components that handle all things related to the generation of FPGA code and hardware. A dedicated Mercurium FPGA phase extracts the code of the accelerated task and generates an HLS wrapper that manages data communication. An external tool, autoVivado, works coupled with Mercurium to synthesize the generated wrappers and design and generate the hardware bitstream.

The autoVivado tool is also used to include some advanced features on the bitstream, such as support for hardware instrumentation and the ad-

dition of the *Asynchronous Task Manager* and the *Task Batch Manager*, both detailed in Chapter 5.

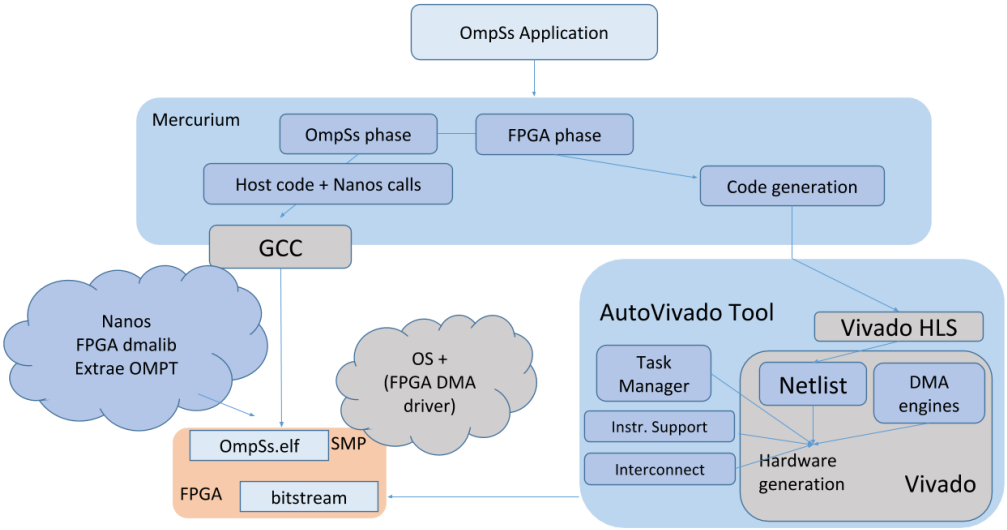


Figure 6.1: OmpSs@FPGA compilation flow

Transparent handling of CPU/FPGA communication required some modifications in the Nanos++ runtime system.

6.1 Mercurium modifications

Mercurium compiler has been modified to include a phase to handle the FPGA part of an OmpSs@FPGA application.

This Mercurium phase extracts the code of each FPGA task into its own source file, including an autogenerated HLS wrapper that will become the top function of the IP core of the accelerator (Vivado HLS project), which will be later compiled by Xilinx Vivado HLS.

6.1.1 Wrapper generation

The autogenerated wrapper acts as an interconnection between the accelerated task and the CPU/memory. Communication with the CPU is done through two types of interfaces: two 64-bit AXI-Stream ports (an input and an output), and a 64-bit AXI port for each data dependence.

Code snippet 6.1 shows a simple code with two dependencies with `copy_in` and `copy_out` clauses. In this code, there are a few HLS directives, in magenta, inserted by the programmer, to optimize the code.

```
1  #pragma omp target device(fpga,smp) copy_deps onto(0,1)
2  #pragma omp task in(v_a[0:SIZE-1], v_b[0:SIZE-1]) out(v_c[0:SIZE-1])
3  void vector_mult(float *v_a, float *v_b, float *v_c) {
4      int i;
5
6      #pragma HLS array_partition variable=v_a block factor=SIZE/2
7      #pragma HLS array_partition variable=v_b block factor=SIZE/2
8
9      #pragma HLS pipeline
10     for (i = 0; i < SIZE; i++) {
11         v_c[i] = v_a[i]*v_b[i];
12     }
13 }
14
15 int main() {
16     float A[SIZE];
17     float B[SIZE];
18     float C[SIZE];
19     initialize_vector(A, SIZE);
20     initialize_vector(B, SIZE);
21     vector_mult(A, B, C);
22     #pragma omp taskwait
23 }
```

Code snippet 6.1: OmpSs@FPGA code with Vivado HLS directives

Code snippet 6.2 shows the interface and protocol automatically generated by the OmpSs@FPGA ecosystem. This is part of the wrapper generated.

```
1  void vector_mult_hls_automatic_mcxx_wrapper(
2       hls::stream<axiData> &inStream,
3       hls::stream<axiData> &outStream,
4       ap_uint<32> accID,
5       float *mcxx_v_a,
6       float *mcxx_v_b,
7       float *mcxx_v_c) {
8  #pragma HLS INTERFACE ap_ctrl_none port=return
9  #pragma HLS INTERFACE axis port=inStream
10 #pragma HLS INTERFACE axis port=outStream
11 #pragma HLS INTERFACE m_axi port=mcxx_v_a
12 #pragma HLS INTERFACE m_axi port=mcxx_v_b
13 #pragma HLS INTERFACE m_axi port=mcxx_v_c
```

Code snippet 6.2: OmpSs@FPGA HLS wrapper header

A minimum of four ports are defined in each wrapper: the two 64-bit AXI-Stream ports, *inStream* and *outStream*, used to receive the memory addresses of the data dependencies and to inform of the finalization of the task execution, respectively; a 32-bit unsigned integer *accID*, which is the global identifier of the accelerator; and a 64-bit AXI port for each data dependence.

Code snippet 6.3 shows the part of the wrapper function that receives the header of the task info struct (Figure 5.6), described in Chapter 5.

```
1      uint64_t __accHeader;
2      uint32_t __destID, __compute;
3
4      __accHeader = inStream.read().data;
5      __compute = __accHeader;
6      __destID = __accHeader>>32;
```

Code snippet 6.3: HLS wrapper reads the task info header

The task info header contains the *compute* flag that indicates if the computation part of the accelerator has to be executed or not and *destID*, the identifier of the destination where to send the *finished* signal once the accelerator finishes execution.

Once the task info header is read, the wrapper proceeds to read the task argument information. Figure 6.4 shows the part of the wrapper that performs the reception of that information. A loop with as much iterations

as dependencies with `copy` clauses is automatically generated. Inside the loop, the information regarding the argument is read from the input AXI-Stream port.

If the argument represents an input dependence, either `in` or `inout`, the argument is copied right away from memory with a *memcpy* to the local variable. On the other hand, if the argument is an output dependence, either `out` or `inout`, the argument information (cached flag and memory address) is stored in a temporary buffer to be used once the computation has taken place.

Code snippet 6.4 shows the code produced when compiling Code snippet 6.1. Accelerator arguments *v_a* and *v_b* represent `in` dependencies and, as such, are copied from kernel space through their corresponding AXI port (*mcxx_v_a* and *mcxx_v_b*). Argument *v_c* corresponds to an `out` dependence and so its information is stored in temporary buffers.

```
1   uint64_t __cached_id_out[1], __argAddr_out[1];
2   uint64_t __cached_id, __argAddr;
3   uint32_t __argCached, __argID;
4
5   for (__i = 0; __i < 3; __i++) {
6     __cached_id = inStream.read().data;
7     __argCached = __cached_id;
8     __argID = __cached_id >> 32;
9     __argAddr = inStream.read().data;
10    switch (__argID) {
11      case 0:
12        memcpy(v_a, (float *)(mcxx_v_a + __argAddr), ...);
13        break;
14      case 1:
15        memcpy(v_b, (float *)(mcxx_v_b + __argAddr), ...);
16        break;
17      case 2:
18        __cached_id_out[0] = __cached_id;
19        __argAddr_out[0] = __argAddr;
20        break;
21      default:;
22    }
23  }
```

Code snippet 6.4: OmpSs@FPGA HLS wrapper arguments reading

The code of the accelerated task is copied as is in the HLS code. Before

calling the function, the wrapper checks if the *compute* flag is enabled or not (Code snippet 6.5).

```
1     if (__compute)
2         vector_mult(v_a, v_b, v_c);
```

Code snippet 6.5: OmpSs@FPGA HLS wrapper computation

Once the function has been executed, and with it the computation part of the task, the wrapper writes back to kernel space the variables representing an output dependence (`out` or `inout`). It reads the *cached* flag and the memory address from the temporary buffers where they have been previously stored and proceeds to copy the data with a *memcpy* through the given AXI port.

In Code snippet 6.6, the wrapper is copying back to the CPU memory the third argument of the task, *v_c*, which is the vector resultant of the multiplication of the two input vectors.

```
1     for (__i = 0; __i < 1; __i++) {
2       __cached_id = __cached_id_out[__i];
3       __argCached = __cached_id;
4       __argID = __cached_id >> 32;
5       __argAddr = __argAddr_out[__i];
6       switch (__arg_id) {
7         case 2:
8           memcpy(mcxx_v_c + __argAddr, (float *)v_c, ...);
9           break;
10        default:;
11      }
12    }
```

Code snippet 6.6: OmpSs@FPGA HLS wrapper writes out dependencies

To finish execution, the wrapper sends the *finished* signal, consisting of the accelerator identifier *accID*, to the destination specified by *destID* (Code snippet 6.7).

```
1    axiData __output = {0,0,0,0,0,0,0};
2    __output.keep = 0xFF;
3    __output.data = accID;
4    __output.dest = __destID;
5    __output.last = 1;
6    outStream.write(__output);
```

Code snippet 6.7: OmpSs@FPGA HLS wrapper sends *finished* signal

The destination signal *destID* is used to guide the interconnection within the FPGA and can be used to inform different elements of the completion of the task. It is usually used to inform the CPU that the task has finished, but it can also be used to inform the *Task Batch Manager* that a task within a batch has finished, or inform the *Asynchronous Task Manager*, by sending the signal to the *Finished Task Manager*, that an asynchronous task has been executed.

If the hardware support for instrumentation is enabled in the compilation, Mercurium compiler would add additional code to annotate the main parts of the accelerator.

The accelerator would receive two extra arguments in the task info header, *instrCounterAddr* and *instrBufferAddr*, the addresses of the hardware counter and the buffer to where store the instrumentation data, respectively.

Additionally, the compiler would include sampling of the hardware counter before and after each one of the three parts the accelerator is divided in: the reception of arguments, the computation and the writing of the output dependencies.

Right after the sampling the hardware counter after the output dependencies writing, the samples are written in the hardware instrumentation buffer, so the CPU can read them.

Code snippet 6.8 shows a generic example of an instrumented accelerator.

```
1   counter_t  __counter_reg[4];
2   uint64_t __instrCounterAddr, __instrBufferAddr;
3
4   __instrCounterAddr = inStream.read().data;
5   __instrBufferAddr = inStream.read().data;
6
7   // Hardware counter sample 0
8   memcpy(&__counter_reg[0], __instrCounterAddr, ...);
9   /*
10   * Read arguments and input dependencies
11   */
12  // Hardware counter sample 1
13  memcpy(&__counter_reg[1], __instrCounterAddr, ...);
14  /*
15   * Compute
16   */
17  // Hardware counter sample 2
18  memcpy(&__counter_reg[2], __instrCounterAddr, ...);
19  /*
20   * Write output dependencies
21   */
22  // Hardware counter sample 3
23  memcpy(&__counter_reg[3], __instrCounterAddr, ...);
24
25  // Write samples to hardware instrumentation buffer
26  memcpy(__instrBufferAddr, __counter_reg, ...);
27  /*
28   * Send finished signal
29   */
```

Code snippet 6.8: OmpSs@FPGA HLS wrapper hardware instrumentation

6.1.2 New Compier and Linker Flags

The new `fpga` Mercurium version, alias `fpgacc`, includes new compiler and linker flags to support the FPGA bitstream generation.

The compilation flag `--variable=bitstream_generation:ON` makes `fpgacc` compiler generate a Vivado HLS wrapper code for each of the `fpga` tasks (accelerators) appearing in a source code.

The linking flags are used to determine how to generate the bitstream with all the accelerators found among all the source codes with compilation flag `--variable=bitstream_generation:ON`. Those flags are, among

others, the target board of the bitstream generation (e.g. `--board=AXIOM`), the accelerator frequency in MHz (e.g. `--clock=200`), if the accelerators should have instrumentation support (e.g. `--hardware_instrumentation`), the project name (e.g. `--name=MxM`), and the directory where the Vivado HLS and Vivado projects should be generated.

For example, a possible command line to compile and link a program (`program.c`) follows:

```
fpgacc --ompss --variable=bitstream_generation:ON \
    -o program program.c \
    --Wf,"--board=AXIOM,--name=vivado_project_name" \
    --Wf,"--clock=200,--dir=$(VIVADO_WORKSPACE)" \
    --Wf,"--hardware_instrumentation,-v"
```

This will generate a intermediate source code for each FPGA accelerator and will make Mercurium compiler invoke the autoVivado tool to generate the bitstream of the accelerators during the linking process. The autoVivado tool details are explained in the following section.

6.2 autoVivado: Automatic bitstream generation

autoVivado is a tool that has been developed with the objective of automatizing the process of generating a bitstream using the Xilinx toolchain. autoVivado integrates all the synchronization and communication techniques analyzed in this master thesis, that can be activated or not. Indeed, autoVivado is a modular tool that can be easily updated with new board supports and features.

It can be integrated with the Mercurium compiler to produce OmpSs@FPGA compliant bitstreams, or use it as a standalone tool to automatically generate bitstreams from a set of source files.

Given the significant amount of time that a bitstream generation takes, the tool divides the generation in 6 steps, so when an error occur in one of them, the generation can be resumed from the step that failed instead of starting over. The 6 steps the generation is divided in are: HLS, design, synthesis, implementation, bitstream and device tree.

autoVivado tool receives a configuration file input with bitstream information (target board, accelerator frequency, project name, etc.) and other details about the steps to be done of the bitstream generation.

6.2.1 Configuration file

The autoVivado configuration file can either be generated manually or through the `fpgacc-bit.sh` script included in the autoVivado toolchain. Figure 6.2 shows a screenshot of the help command output of autoVivado, which details the set of options to configure the behaviour of autoVivado and the parameters of the bitstream.

```
mvidal@bsc:~/workspaces/BSC/autoVivado/scripts$ ./fpgacc-bit.sh --help

usage: fpgacc-bit.sh options

This script generates the configuration file needed to run autoVivado and executes the tool with the given parameters.

OPTIONS:
-h, --help                  Show this message
-n, --name                  Project name
-d, --dir                   Path where the project directory tree will be created
-b, --board                 Board model. Currently supported: Zynq706 Zynq702 Zedboard AXIOM Ikergune
-c, --clock                 FPGA clock frequency in MHz. Valid range: 10-250 MHz.
--taskGroup                 Adds hardware support for TaskGroups
--async                     Adds hardware support for asynchronous submits
--hardware_instrumentation  Adds support for hardware instrumentation
--disable_IP_caching        Disables IP caching. Significantly increases compilation time
--IP_cache_location         Path where the IP cache will be located. If no path is set, the cache will be created in the autoVivado folder
--from_step                 Initial compilation step. Compilation steps: HLS design synthesis implementation bitstream devicetree
--to_step                   Final compilation step. Compilation steps: HLS design synthesis implementation bitstream devicetree
-v, --verbose               Prints Vivado messages
```

Figure 6.2: Screenshot of autoVivado `help` command output with the complete set of options to configure the tool

Mercurium uses the latter option and runs the script right after generating the wrappers for all the FPGA-accelerated tasks, during the linking step of the Mercurium compilation.

Code snippet 6.9 shows the configuration file used to generate the bitstream of the vector multiplication example code. This configuration file was automatically generated by Mercurium during its compilation to bitstream.

The minimum information needed to generate the bitstream is the model of the board (`BOARD_PART`, line 1), the frequency of the FPGA clock (expressed as the period, `CLOCK_PERIOD`, line 4) and the list of accelerators to include (`KERNELS`, lines 7-13, with 5 accelerators).

Additionally, the file includes parameters to configure the behaviour of autoVivado, for example the directory and name of the autoVivado project (`PROJECT_DIR` and `PROJECT_NAME`, lines 2 and 3), the initial and final

generation steps that has to perform the tool (`FROM_STEP` and `TO_STEP`, lines 5 and 6) and which advanced features are enabled (lines 15-17).

```
1  BOARD_PART=xc7z045ffg900-2          #Board part
2  PROJECT_DIR=/path/to/autoVivado/project/ #Project dir
3  PROJECT_NAME=vector_mult            #Project name
4  CLOCK_PERIOD=10                     #Clock period (ns)
5  FROM_STEP=0                         #Initial generation step
6  TO_STEP=4                           #Final generation step
7  KERNELS="
8  [0]=/path/to/accel/0:2:vector_mult_hls_automatic_mcxx.cpp
9  [1]=/path/to/autoVivado/HLS/src/Hardware_Counter.cpp
10 [2]=/path/to/autoVivado/HLS/src/Task_Batch_Manager.cpp
11 [3]=/path/to/autoVivado/HLS/src/Ready_Task_Manager.cpp
12 [4]=/path/to/autoVivado/HLS/src/Finished_Task_Manager.cpp
13 "
14 NUM_KERNELS=1                       #Number of kernels
15 HARDWARE_INSTRUMENTATION=true
16 ASYNC_TASK_MANAGER=true
17 TASK_BATCH_MANAGER=true
```

Code snippet 6.9: autoVivado configuration file

autoVivado main script (`autoVivado.sh`) parses the configuration file and proceeds to perform the generation steps specified in the file. It can perform just one step or all of them, but always in a sequential way. If the programmer tries to run autoVivado from the synthesis step to the bitstream step without passing through steps HLS and design, it will most likely fail unless the programmer has manually performed them and conditions to start the synthesis step are met.

It is possible, however, to perform middle steps as long as a previous autoVivado run has completed the initial steps correctly.

6.2.2 Generation step 0: HLS

In the first step of bitstream generation, autoVivado parses the input configuration file and obtains the list of accelerators that need to be included in the bitstream.

Accelerators file names have to follow a certain pattern in order for autoVivado to correctly handle them. The identifier of the accelerator, as well as the number of instances of that accelerator, have to be prepended to the file

name, separated by a colon (:). For example, the name of the vector multiplication accelerator is `0:2:vector_mult_hls_automatic_mcxx.cpp`, meaning that the identifier of the accelerator is 0 and the bitstream will contain 2 instances of that IP core. The suffix `_hls_automatic_mcxx` is appended by Mercurium to identify the source files that have been automatically generated.

autoVivado creates a Xilinx Vivado HLS project for each of the accelerators (hardware managers and instrumentation counter included) and proceeds to compile them source-to-source from C/C++ to Verilog/VHDL.

The accelerators are compiled using the target clock period and target FPGA board that are specified in the configuration file.

After the correct source-to-source compilation of each accelerator, autoVivado checks the reports and obtains the FPGA resource utilization estimation that Vivado HLS computes. autoVivado maintains an estimation of the total resource utilization and aborts the bitstream generation if it founds that the set of accelerators does not fit into the physical limitations of the board.

Correctly compiled accelerators are from now on considered IP cores and are added into a local IP repository that will be used in following generation steps by Xilinx Vivado.

6.2.3 Generation step 1: Design

On the second step, the actual design of the bitstream takes place. In this step, autoVivado generates a design with all the IP cores compiled in the first step and generates the interconnection.

It takes advantage of the Xilinx Vivado ability to run Tool Command Language (Tcl) scripts to encapsulate the main parts of the design in Tcl *templates*.

A base design template contains the IP core that represents the CPU already configured, and with all the PS/PL communication interfaces enabled. There is a template for each CPU family (currently 32-bit Zynq and 64-bit Zynq Ultrascale+).

Another template encapsulates all the elements that surround each accelerator (DMA, interconnection, clock and reset signals...) already configured and connected between them. A placeholder dummy accelerator IP is present in the template, which is swapped by the real IP once the template is placed in the design.

Additional templates encapsulate the *Asynchronous Task Manager* (Figure 5.7), the *Task Batch Manager* and the infrastructure to support hardware instrumentation.

autoVivado creates a Xilinx Vivado project in the path specified by the `PROJECT_DIR` configuration option and uses the different templates to instantiate the IPs.

Communication between IPs and the DDR memory is done through AXI ports. Standard AXI interconnect IPs are placed at each PS/PL interface to act as multiplexers to allow multiple connections to each port.

On the other hand, AXI-Stream ports are used to send and receive information between accelerator and manager IPs through standard AXI-Stream interconnect IPs. In these interconnects is where the *destID* argument that every accelerator receives in the task header (Figure 5.6) comes into play.

AXI-Stream interconnect IPs contain an internal crossbar that can be configured to distribute the data to different destinations with the *TDEST* signal. The crossbars are configured by autoVivado to distribute the data the following way:

- *TDEST* $0x00 - 0x0F$: Data is sent to the accelerator identified with the same value as the *TDEST* signal.
- *TDEST* $0x10$: Data is sent to the *Task Batch Manager*.
- *TDEST* $0x11$: Data is sent to the *Finished Task Manager*.
- *TDEST* $0x1E$: Data is sent back to the sender.
- *TDEST* $0x1F$: Data is sent to the CPU.

After all the IPs have been placed and interconnected in the design, autoVivado maps the DMAs and BRAMs to the CPU address space and validates the design generated.

6.2.4 Generation step 2: Synthesis

Although being the step that takes most time, it is one of the simplest ones from the autoVivado point of view.

In this step, autoVivado runs the Xilinx Vivado synthesis command to transform the design generated in step 2, specified in a HDL language, either Verilog or VHDL, into a gate-level representation.

It uses some of the parameters provided to autoVivado, such as the target device and the FPGA clock frequency.

6.2.5 Generation step 3: Implementation

Similarly to the previous one, this step is the second longest but with almost no actions taken by autoVivado.

In this case, autoVivado runs the implementation command of Xilinx Vivado to transform the gate-level representation of step 3 into a placed and routed design, ready for bitstream generation.

The implementation process runs several iterations of placing, routing and optimization sub-processes until all the requirements are met.

6.2.6 Generation step 4: Bitstream

In step 5, the bitstream is generated from the placed and routed design of step 4. The Hardware Description File, which contains information of the hardware design, is also generated.

Bitstream file is copied by autoVivado to the path specified by `PROJECT_DIR` to make it easier for the programmer to find it, since it is usually buried in several levels of folders inside the Vivado project.

6.2.7 Generation step 5: Device tree

As the final step, autoVivado generates the device tree using the *hsi* tool provided by the Xilinx Vivado SDK. This tool takes the Hardware Description File of the previous step and generates device tree sources with the characteristics of the bitstream.

The device tree sources are modified to include a device used by the DMA user library to interact with the different DMAs in the bitstream.

The device tree is used by the Linux operating system to build the */dev/* directory, which is where the DMAs are exposed to the operating system as well as a special device to reprogram the FPGA with a new bitstream.

6.3 Runtime modifications

We wrote external library containing all the functions related to FPGA tasks, called *xTasks*, to simplify the implementation of the FPGA plugin of Nanos++ runtime. The library already implemented the *synchronous data transfer offloading* technique. This way, we could easily test the different techniques by rewriting some of those functions and just recompile the library.

6.3.1 Asynchronous submissions of ready tasks

To implement the asynchronous submission of ready tasks, the task creation, task submit and task wait functions needed to be modified.

The function in charge of the creation of tasks was modified to, besides creating and initializing the task information struct (Figure 5.6), also initialize the struct representing an asynchronous ready task (Figure 5.9).

Arguments are added to the task information struct, with no extra action needed.

At submit time, the asynchronous ready task struct containing the pointer to the task information struct is stored in the *readyQueue* BRAM, which has been previously mapped to memory. The *valid* field of the asynchronous ready task struct is set, so the *Asynchronous Task Manager* will read the *readyQueue* element and proceed to submit the task to the corresponding accelerator.

Finally, to wait for an asynchronous task, the *xTasks* wait task function required to be completely rewritten. Instead of waiting for the *finished* signal sent through a DMA transfer, the function implements an active polling looking for a valid element on the *finishedQueue* BRAM representing a finished task. When found, the finished task is returned to Nanos++

and the *finishedQueue* slot is freed.

6.3.2 Dynamic creation of task batch and submission

To correctly create and submit a task batch, we need to generate the structure of Figure 5.14. Thus, we need to create a meta-task, representing the task batch structure, which contains all the tasks. Inner tasks are created normally, however they must be stored inside the task batch struct and their submission to the FPGA must not take place.

The batch region is confined within two *xTasks* functions. The first function, which marks the start of the region, allocates the task batch struct and initializes the elements. The second function, which marks the end of the batch region, submits the batch to the hardware manager waits for the *finished* signal and frees the task batch buffer. Both DMA transfers are blocking, so there is an implicit `taskwait` at the end of the batch region.

To create the task batch, we disable the dependencies computation of the task batch region, to prevent Nanos++ from scheduling tasks that are part of the batch, and we fool Nanos++ by informing it that the tasks are being executed when they are actually being inserted into the batch. Once Nanos++ has created all the tasks of the batch, we submit it to the hardware manager.

Our intention, however, is to actually use the `task batch` directive and modify Nanos++ to call the proper task creation function, not execute the task submit function at every inner task and submit the entire batch at the end of the region.

Chapter 7

Evaluation

In this chapter, we present the results of the experiments performed to evaluate the different communication and synchronization techniques proposed in Chapter 5.

7.1 Experimental setup

Experiments were performed using a single Xilinx Zynq 706 FPGA board running a Linaro 14.04 Linux distribution with a 4.6.0 kernel.

The board, already detailed in Section 2.1.1, consists of a System-on-Chip including a dual-core ARM Cortex-A9 running at 667MHz and a Xilinx Kintex-7 FPGA running at 100MHz.

Given that the OmpSs programming model ecosystem for FPGA is still in development, and many of its components were developed concurrently with this master thesis, still there is not a full set of applications to test our modifications.

In order to evaluate the benefits of the new synchronization and communication techniques, matrix multiply, a heavily used kernel in scientific applications, has been analyzed. Matrix multiply is used in Cholesky decomposition, *k*-means clustering, convolutional neural networks, etc.

We used the same blocked matrix multiplication base code in all our experiments, only modifying the code surrounding it. Four different versions were developed to evaluate the contributions of this master thesis:

- **AXI-Stream** based data transfers **standalone** version. The HLS code,

written by hand, includes the reading of the input matrices and the writing of the output ones, using the AXI-Stream protocol. On the host side, data handling and calls to the communication DMA library were also coded by hand.

- **AXI-Stream** based data transfers **OmpSs** version. Same HLS code as the previous, no modifications. The host code is generated by Mercurium, including calls to the Nanos++ runtime and the DMA library. A couple of functions to be able to send whole data through DMA transfers have been coded in the DMA user library. And the Nanos++ runtime has been slightly modified in order to use these functions when submitting OmpSs tasks to the FPGA.

- Data transfer **offload standalone** version. Mercurium generated HLS code and bitstream designed and generated by autoVivado. The host part is manually coded to include calls to the unmodified DMA library.

- Data transfer **offload OmpSs** version. This version utilizes the full OmpSs@FPGA toolchain. Mercurium generates both the HLS and the host code and autoVivado automatically designs and generates the bitstream. The host code takes advantage of the Nanos++ runtime.

Given the physical limitations of the FPGA board used, we generated three types of bitstreams defining three different FPGA environments:

- small-size matrix multiply IP core with a block size of 32x32.

- medium-size matrix multiply IP core with a block size of 64x64.

- large-size matrix multiply IP core with a block size of 128x128.

We divided the evaluation experiments in two categories: standalone and OmpSs.

The experiments of the standalone category were done to evaluate the performance of the new communication and synchronization techniques on their own, without any OmpSs-related overhead. Therefore the Nanos++ runtime system is not used and the computation is done in a single accelerator.

OmpSs experiments were done to evaluate the impact on the performance

of the initial implementations of the new techniques.

To test the batching technique, we batched the innermost loop of the set of nested loops that handles the matrix blocks. This way, we can offload the computation of an entire block of the result matrix *C* and reduce the number of DMA transfers from N^3 to N^2, being N the number of blocks of a side of the matrix.

Moreover, we can take advantage of the *argCached* flag and avoid unnecessary data copies of the matrix *C* block, which is shared among all tasks of the batch.

7.2 Performance analysis

For each combination of technique, block size and matrix size, we run a total of 10 executions and computed the arithmetic average. For each execution, we measured the elapsed time it took to perform the whole matrix multiplication, from the sending of the input matrices to the receiving of the result matrix.

The performance results are presented through a series of charts, each of whom represents an experiment performed with a certain accelerator block size, stated above the chart, and using the different communication techniques:

- *AXI-Stream* are the results when using the AXI-Stream based communication system that was used at the beginning of this master thesis.
- *Offload* uses the transfer offloading technique presented in Section 5.1.
- *Asynchronous* uses the asynchronous task management technique presented in Section 5.2.
- *Task batch* uses the batching technique presented in Section 5.3 without exploiting the accelerator internal cache.
- *Task batch + Cached* also uses the batching technique, but marking the output matrix of all but the first and last tasks of the batch as cached.

7.2.1 Small-sized accelerators

Results from small-sized accelerators show a significant improvement, both in standalone and in OmpSs. This is due to the overhead of creating and submitting each of the DMA transfers in *AXI-Stream*, which, given the small size of the accelerator, is comparable to the cost of receiving the data and performing the computation. This leads the accelerator to spend a lot of time being idle, waiting for each DMA submit.

Offload overcomes this problem performing a single small DMA transfer, with the information of the copies so the accelerator can copy all the data right away, without having to wait between copies. Figure 7.1 shows the time reduction when using this technique in the standalone application. We can also see that the rest of the techniques, that derive from *Offload*, further reduce the execution time.

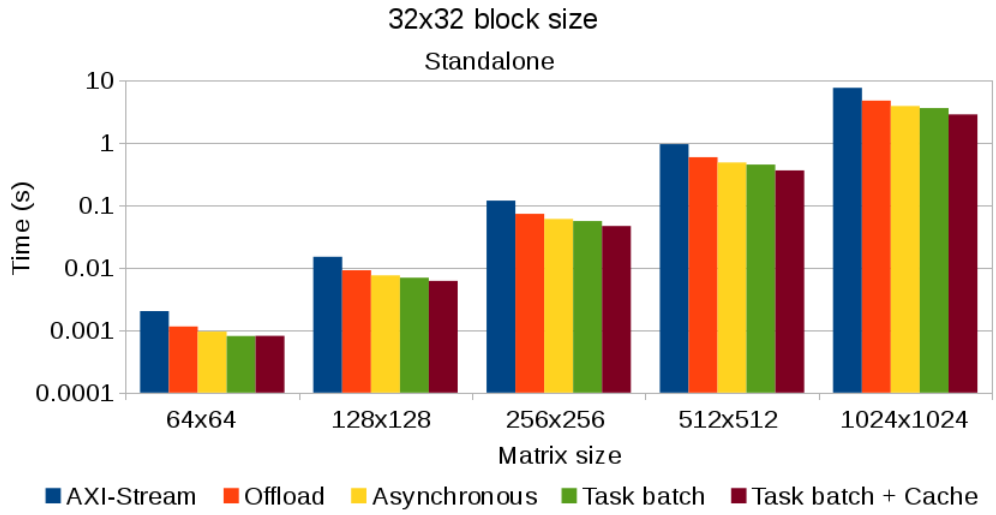


Figure 7.1: Standalone performance of the communication techniques in small-sized accelerators

Specially effective reducing the execution time are the batching techniques *Task batch* and *Task batch + Cache*. Having a multitude of small tasks to execute introduces a lot of idling time in the accelerator while it waits for the next task to arrive. If we batch a set of tasks and send them in a batch, we allow the accelerator to execute them one after the other, with no idling time between them. It is the same idea that led to the development of the *Offload* technique at dependency level, but applied at a task level.

Figure 7.2 shows the different speedups obtained in this standalone context.

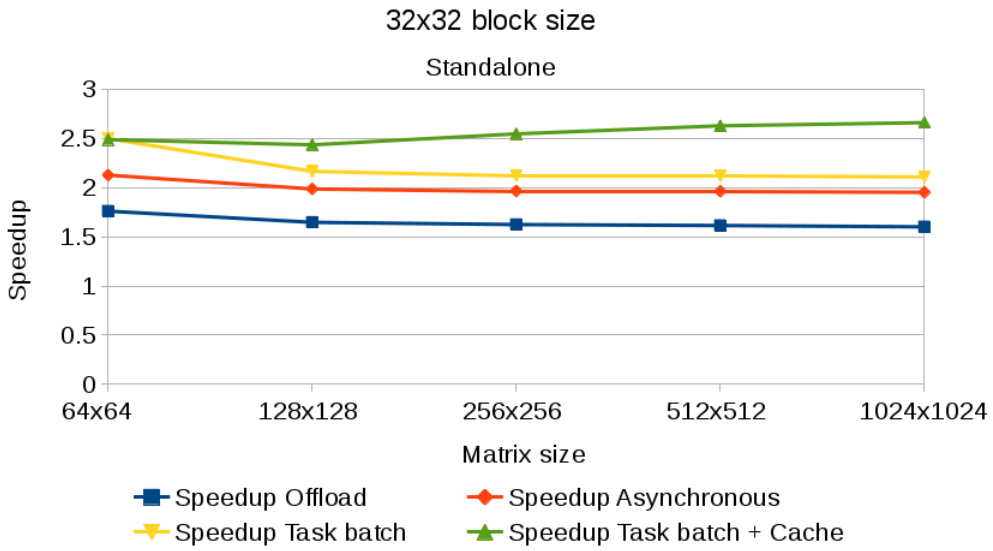


Figure 7.2: Standalone speedup of the communication techniques against AXI-Stream communication in small-sized accelerators

We can observe that the speedup of the *Task batch + Cache* technique increases with the matrix size. This is due to the increasing amount of data we avoid copying back and forth the CPU.

Figure 7.1 summarizes the communication savings of using the *cache* flag to exploit the accelerator internal memory. Several MBs of data copies can be avoided if we do not perform the copies regarding the *C* matrix in the middle tasks of the batch.

	64x64	**128x128**	**256x256**	**512x512**	**1024x1024**
Number of tasks	8	64	512	4096	32768
Total number of transfers	24+8	192+64	1536+512	12288+4096	98304+32768
Number of transfers avoided	0+0	32+32	384+384	3584+3584	30720+30720
Communication savings	0	256kB	3MB	28MB	240MB

Table 7.1: Summary table of the savings in communication (input+output) when the *argCached* flag is used in small-sized accelerators.

The *argCached* flag used at the moment has only two meanings: a variable can either be cached or not. However, for `inout` variables, like the

C matrix in our application, we could add two extra meanings to the flag: caching the input value or the output value. This way, an additional 8MB of data (4MB input + 4MB output) would not be copied in the 1024x1024 matrix case.

Figure 7.3 shows the execution time of the application when using the full OmpSs@FPGA ecosystem. The OmpSs execution times are, in average, higher than in standalone due to the overhead introduced by the Nanos++ runtime and we are sequentially executing the MxM task in the accelerator.

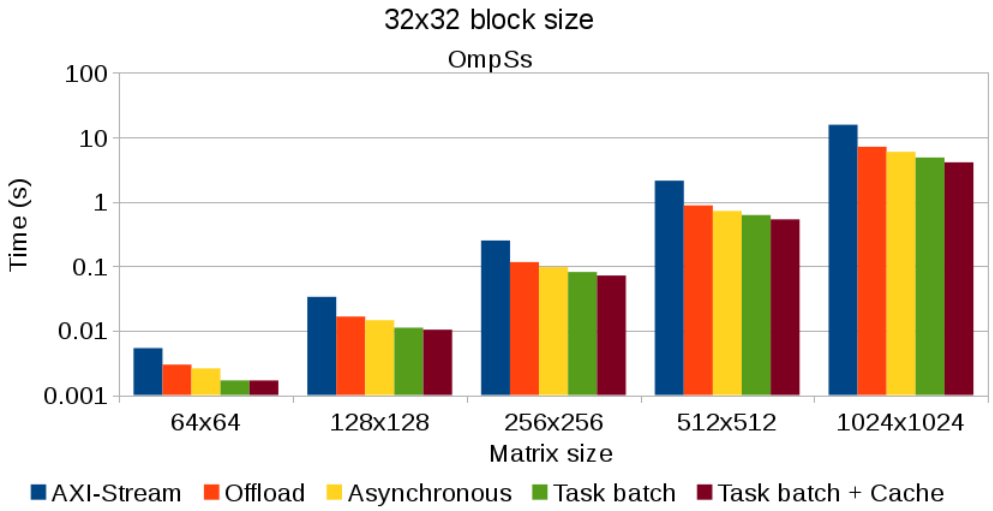


Figure 7.3: OmpSs performance of the communication techniques in small-sized accelerators

However, the techniques proposed help reduce the Nanos++ overhead which leads to higher speedups, shown in Figure 7.4.

In particular, *Task batch* and *Task batch + Cache* benefit from the fact that the tasks of the batch do not need no be inserted into the dependency graph. This yields an additional speedup when there are few tasks to execute, nevertheless, when the number of tasks increases, the runtime can overlap the execution of tasks in the accelerator with the execution of runtime duties, such as inserting tasks into the dependency graph. So, this gain decreases with the number of tasks.

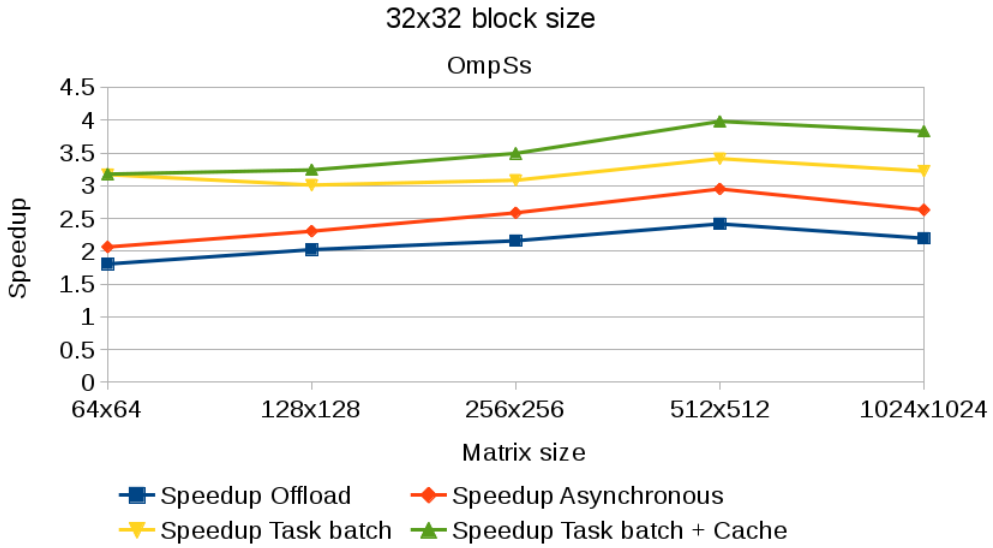


Figure 7.4: OmpSs speedup of the communication techniques against AXI-Stream communication in small-sized accelerators

Finally, Figure 7.5 shows the speedups of *Asynchronous*, *Task Batch* and *Task Batch + Cache* (extensions of the *Offload*) considering *Offload* the base technique. There is more than 15% speedup for the *Asynchronous* extension and more than $1.5\times$ speedup with the *Task batch* and *Task batch + Cache* extensions.

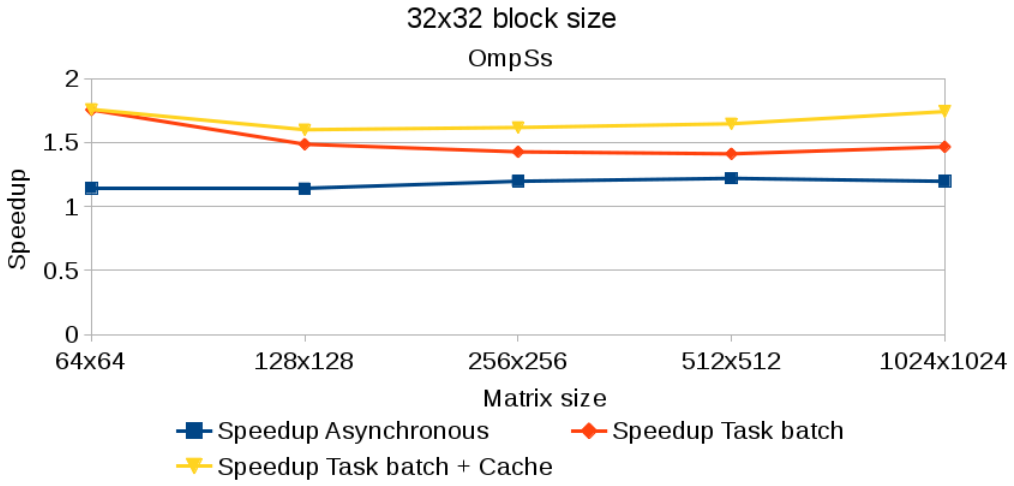


Figure 7.5: Speedup of the new techniques against *Offload* in small-sized accelerators

7.2.2 Medium-sized accelerators

For the medium-sized accelerators, we see a reduction of the benefits of the new techniques, mainly because the weight of the computation part has increased. Figure 7.6 shows the average execution times of the techniques when dealing with different matrix sizes.

The price we had to pay in small-sized accelerators when submitting DMA transfers in *AXI-Stream* has almost disappeared here, because the weight of the computation and communication has increased, whereas the overhead of DMA transfer submits remains almost the same.

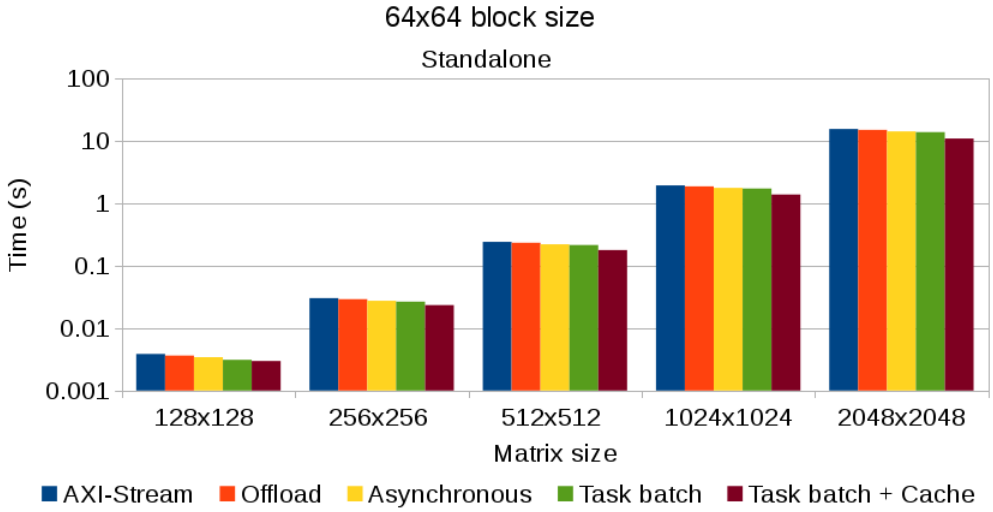


Figure 7.6: Standalone performance of the communication techniques in medium-sized accelerators

If we take a look at the speedup of Figure 7.7, we still can see that the *Offload* technique and its derivatives still yield some degree of speedup compared to *AXI-Stream*. *Offload*, *Asynchronous* and *Task batch* speeds up the application between a 5% and a 25%, specially when there are few tasks to be executed. *Task batch + Cache*, on the other hand, still maintains a good speedup of over 40% thanks to the amount of communication avoided.

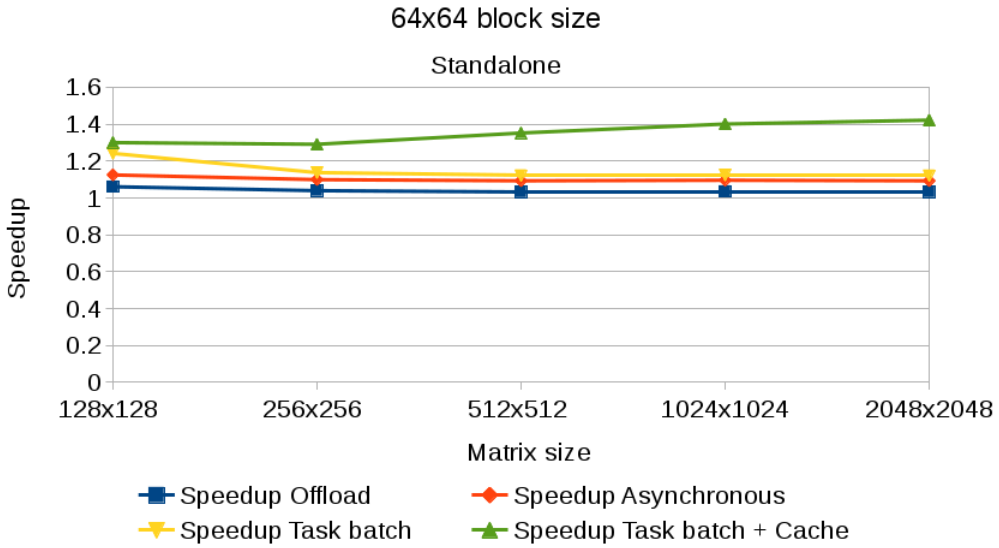


Figure 7.7: Standalone speedup of the communication techniques against AXI-Stream communication in medium-sized accelerators

The table from Figure 7.2 summarizes the total amount of bytes and memory transfers avoided thanks to the *argCached* flag. If we implemented a *argCached* flag for `inout` dependencies to cache either the input or the output value, we could avoid copying an additional 16MB (8MB input + 8MB output) for the 2048x2048 matrix case.

	128x128	**256x256**	**512x512**	**1024x1024**	**2048x2048**
Number of tasks	8	64	512	4096	32768
Total number of transfers	24+8	192+64	1536+512	12288+4096	98304+32768
Number of transfers avoided	0+0	32+32	384+384	3584+3584	30720+30720
Communication savings	0	1MB	12MB	112MB	960MB

Table 7.2: Summary table of the savings in communication (input+output) when the *argCached* flag is used in medium-sized accelerators.

In OmpSs, the reduction in speedup is also significant, but still higher than in standalone due to the runtime overhead. Figure 7.8 shows the execution times of the OmpSs application. In this case, the tasks are large-enough to allow the runtime to submit the DMA transfers before the accelerator finishes the previous execution and to reduce idle time between transfers.

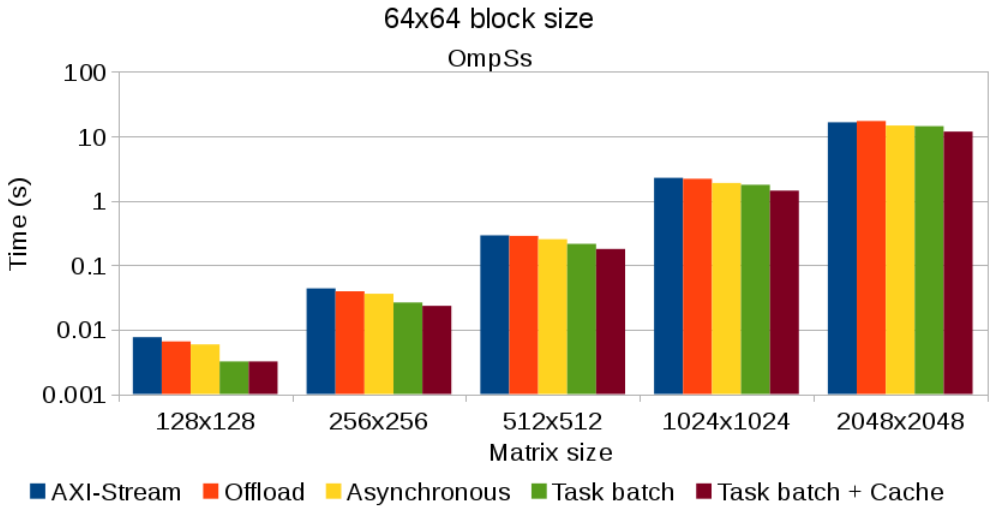


Figure 7.8: OmpSs performance of the communication techniques in medium-sized accelerators

Figure 7.9 shows the speedups obtained running the OmpSs version of the application. As in small-sized accelerators, we can see a peak in speedup in *Task batch* and *Task batch + Cache* when there is a small number of tasks to be executed, attributable to the fact that the runtime does not insert the tasks into the dependency graph. This speedup is reduced the more tasks there are, as the runtime overlaps the execution of accelerator tasks with the handling of the dependency graph.

The speedup gained by exploiting the accelerator internal memory and reducing the number of data copies is still significant, with over 50% of improvement over *AXI-Stream*.

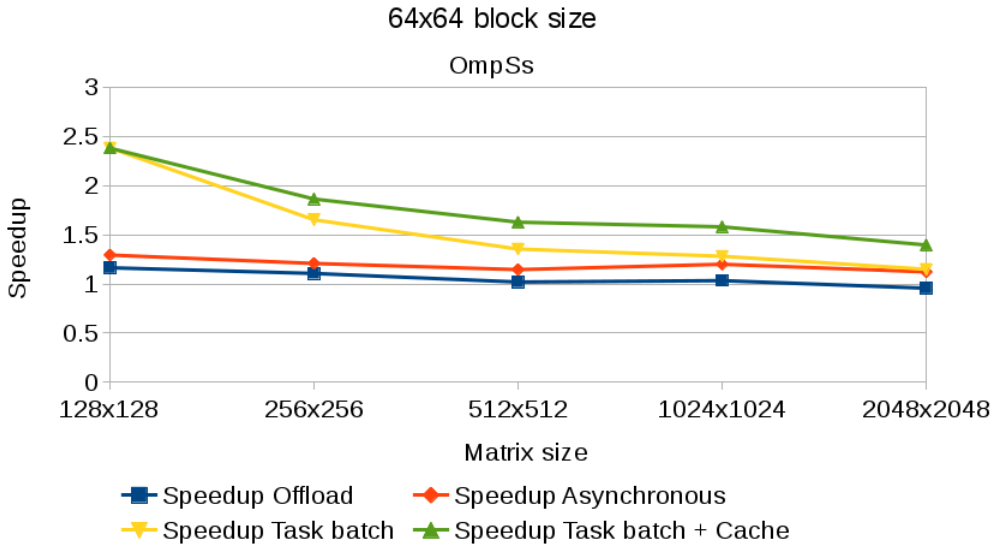


Figure 7.9: OmpSs speedup of the communication techniques against AXI-Stream communication in medium-sized accelerators

Figure 7.10 shows the speedups of the *Offload* extension techniques compared to the *Offload*. We can see that the performance benefits of the extension techniques decreases when the number of tasks to execute are larger. This is due to the same reason we commented in previous section with small-sized accelerators: the overhead of creating tasks and offloading them can be easily overlapped when increasing the number of tasks.

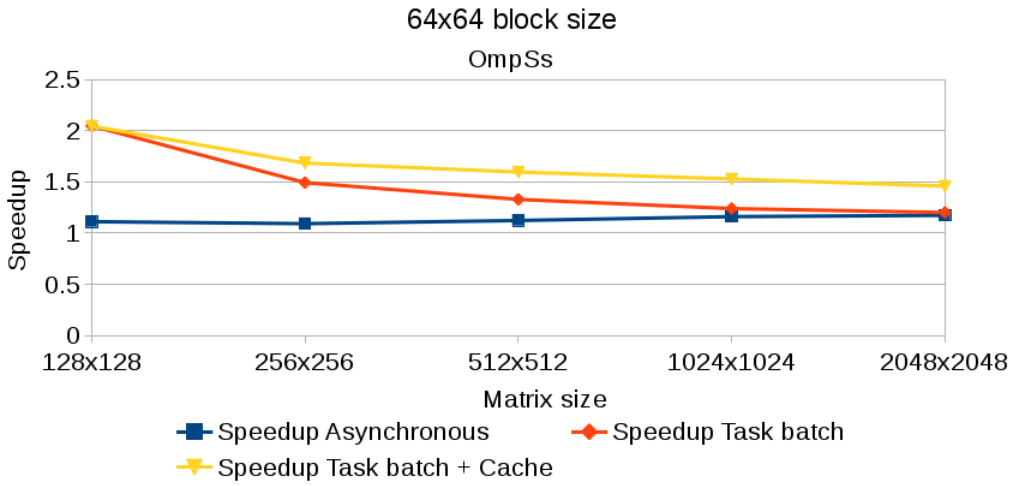


Figure 7.10: Speedup of the new techniques against *Offload* in medium-sized accelerators

7.2.3 Large-sized accelerators

For the case of the standalone experiments, large-sized accelerators seems to overcome the performance of the proposed techniques (Figure 7.11) because the weight of the synchronization with the CPU (DMA submits and idle times between them) is no longer relevant against pure communication and computation. In fact, the overhead introduced to support those techniques, when there is no OmpSs overhead, provokes a light slowdown in performance.

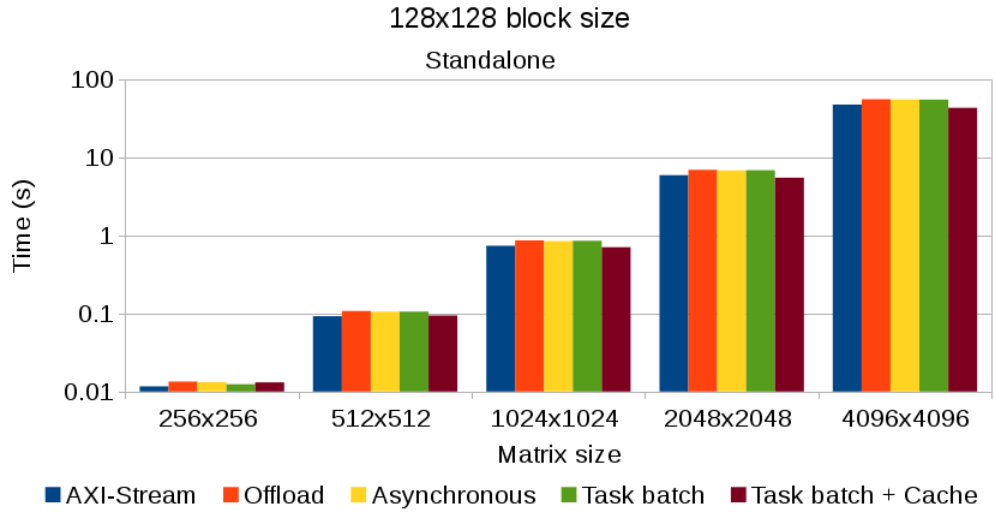


Figure 7.11: Standalone performance of the communication techniques in large-sized accelerators

The only technique that achieves some degree of speedup (Figure 7.12) is *Task batch + Cached* because, apart from reducing synchronization with the CPU, it reduces the actual amount of data communication done at each task execution.

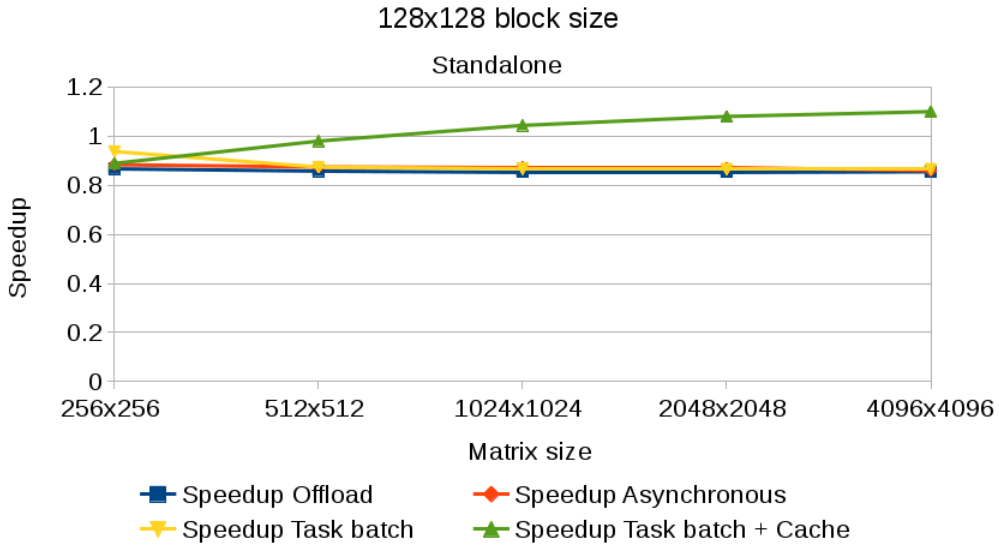


Figure 7.12: Standalone speedup of the communication techniques against AXI-Stream communication in large-sized accelerators

The amount of data communication avoided thanks to the *argCached* flag is summarized in the table of Figure 7.3. Extending the *argCached* flag to support `inout` dependencies, would reduce data communication by an additional 128MB (64MB input + 64MB output) in the 4096x4096 matrix case.

	256x256	**512x512**	**1024x1024**	**2048x2048**	**4096x4096**
Number of tasks	8	64	512	4096	32768
Total number of transfers	24+8	192+64	1536+512	12288+4096	98304+32768
Number transfers avoided	0+0	32+32	384+384	3584+3584	30720+30720
Communication savings	0	4MB	48MB	448MB	3.75GB

Table 7.3: Summary table of the savings in communication (input+output) when the *argCached* flag is used in large-sized accelerators.

For the case of OmpSs applications, the Nanos++ overhead, which specially affects the *AXI-Stream* technique, allows the other techniques obtain better average execution times (Figure 7.13).

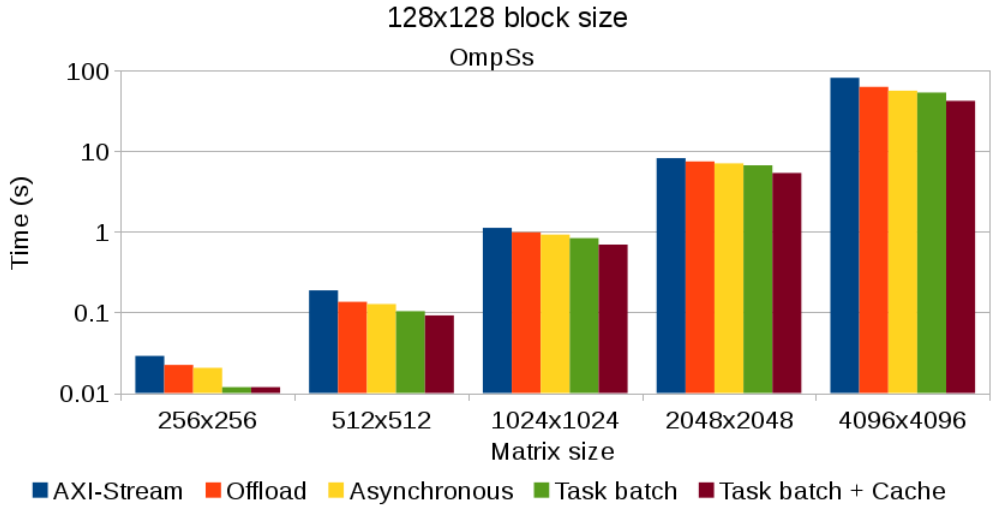


Figure 7.13: OmpSs performance of the communication techniques in large-sized accelerators

Figure 7.14 shows the speedup achieved by the proposed techniques compared to the *AXI-Stream* technique. The trend is the same as with small and medium-sized accelerators: batching techniques have a peak of speedup when the number of tasks to execute is small, and then decreases when the number of tasks increases.

In this large-sized accelerators context, we can also see that there is an increase of the speedup in big matrices, such as 4096x4096, because of an slowdown of the *AXI-Stream* performance. This slowdown seems to be caused by the high number of DMA transfers with heavy data payloads.

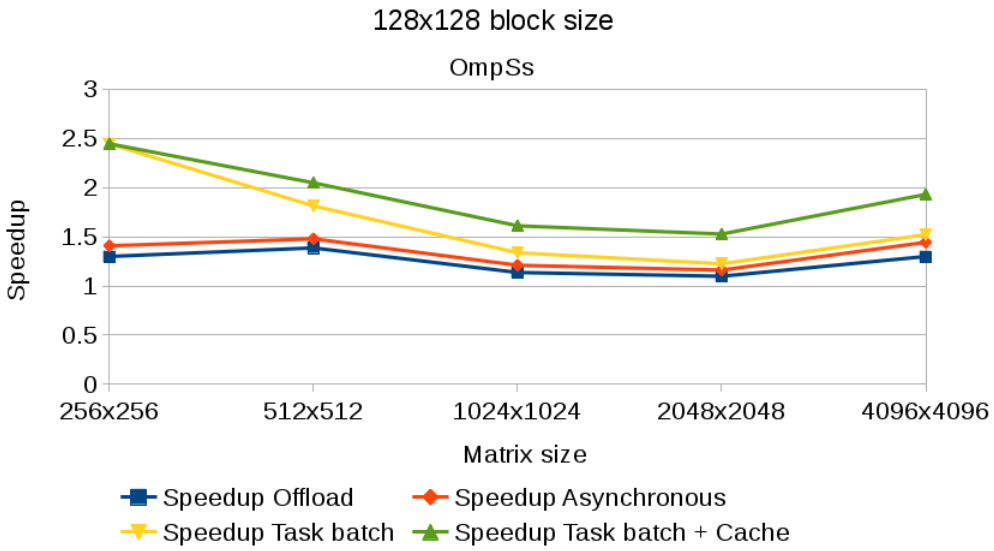


Figure 7.14: OmpSs speedup of the communication techniques against AXI-Stream communication in large-sized accelerators

The previous trend assumption seems to be supported by the numbers of Figure 7.15, where we can see that the relative performance between the proposed techniques is maintained compared to small and medium size accelerators.

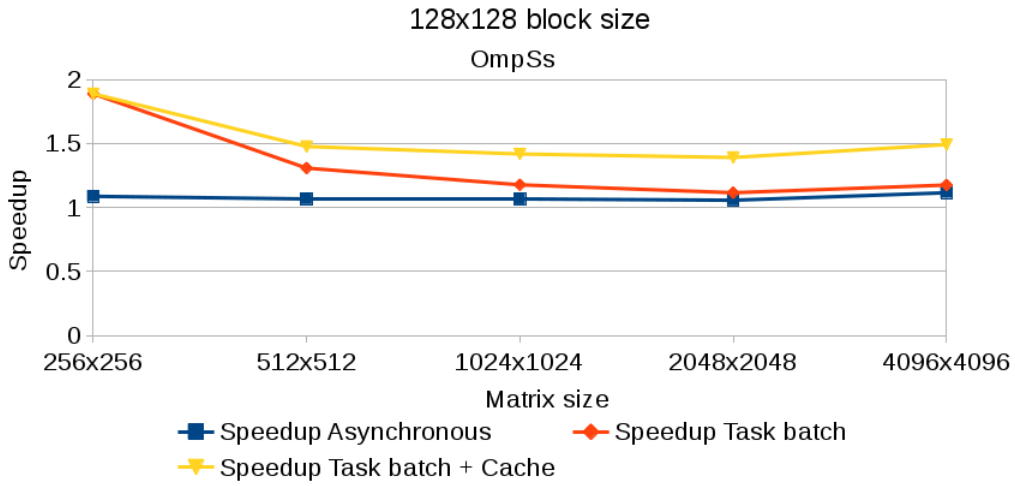


Figure 7.15: Speedup of the new techniques against *Offload* in large-sized accelerators

Chapter 8

Conclusions and Future Work

In this master thesis we propose a set of new communication and synchronization techniques for the FPGA ecosystem of the OmpSs programming model, based on the analysis of the current communication paradigm.

Our proposals include (1) a new communication technique that transfers the master role in communication from the CPU to the FPGA by offloading data transfers to the accelerators; and two techniques to reduce CPU/FPGA synchronization based on (2) asynchronous task management and (3) task batching, which are built over the first proposal.

We also presented the *autoVivado* tool, that automatizes the design and generation of bitstreams. It can be coupled with Mercurium to generate OmpSs-capable FPGA bitstreams and allows the programmer to include the hardware support for the different techniques here proposed.

We have evaluated the performance of our proposals with an extensively used compute kernel: a blocked matrix multiplication. The results obtained show that, under certain conditions, the application performance improved, specially in small-sized accelerators. Particularly, the new techniques improves exploitation of fine-grain parallelism and open the way for data re-using.

Moreover, the second proposal, the asynchronous task management, completely removes explicit synchronization between the CPU and the FPGA and the need of DMA engines and, by extension, drivers, user libraries or modifications to the device tree. We have also seen that the use of DMAs is not an optimal solution for our workloads and might, and at the moment does, carry software bugs.

Part of the contributions of this master thesis have been presented as Proof-of-Concept implementations to evaluate their benefits, while others have only been proposed but not implemented.

The next steps that could be done after this master thesis include:

- Implement the `task batch` OmpSs directive and modify the Nanos++ runtime to accommodate it.
- Explore the capabilities of the *cached* flag and automatize its use in task batches.
- Implement and evaluate the combination of asynchronous task management and task batching techniques.
- Study the necessity of maintaining DMA engines in our hardware designs and eliminate their use from the *xTasks* library and *autoVivado* tool.
- Evaluate the techniques in a multi-accelerator scenario and study how the hardware frequency affects them.

Acronyms

ACP Accelerator Coherency Port.

ASIC Application-Specific Integrated Circuit.

AXI Advanced eXtensible Interface.

DDR Double Data Rate.

DMA Direct Memory Access.

DSP Digital Signal Processing.

FPGA Field Programmable Gate Array.

GP General Purpose.

HDL Hardware Description Language.

HLS High-Level Synthesis.

HP High-Performance.

IP Intellectual Property.

LUT Look-Up Table.

PL Programmable Logic.

PS Processing System.

SoC System-on-Chip.

SRAM Static Random-Access Memory.

Tcl Tool Command Language.

References

[1] Eduard Ayguadé, Rosa M. Badia, Daniel Cabrera, Alejandro Duran, Marc Gonzalez, Francisco Igual, Daniel Jiménez-González, Jesús Labarta, Xavier Martorell, Rafael Mayo, Josep M. Perez, and Enrique S. Quintana-Ortí. A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5568 LNCS, pages 154–167. Springer, Berlin, Heidelberg, 2009.

[2] Antonio Filgueras, Eduard Gil, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Jan Langer, Juanjo Noguera, and Kees Vissers. OmpSs@Zynq all-programmable SoC ecosystem. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays - FPGA '14*, pages 137–146, New York, New York, USA, 2014. ACM Press.

[3] The OpenCL Specification, version 2.2, 2017.

[4] The OpenACC Application Programming Interface, 2013.

[5] Carlos Álvarez, Eduard Ayguadé, Jaume Bosch, Javier Bueno, Artem Cherkashin, Antonio Filgueras, Daniel Jiménez-González, Xavier Martorell, Nacho Navarro, Miquel Vidal, Dimitris Theodoropoulos, Dionisios N. Pnevmatikatos, Davide Catani, David Oro, Carles Fernández, Carlos Segura, Javier Rodríguez, Javier Hernando, Claudio Scordino, Paolo Gai, Pierluigi Passera, Alberto Pomella, Nicola Bettin, Antonio Rizzo, and Roberto Giorgi. The AXIOM software layers. *Microprocessors and Microsystems*, 47:262–277, 2016.

[6] Xubin Tan, Jaume Bosch, Miquel Vidal, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, and Mateo Valero. General Purpose Task-Dependence Management Hardware for Task-Based Dataflow Programming Models. In *2017 IEEE International Parallel*

and Distributed Processing Symposium (IPDPS), pages 244–253. IEEE, may 2017.

[7] Xubin Tan, Jaume Bosch, Miquel Vidal, Carlos Alvarez, Daniel Jimenez-Gonzalez, Eduard Ayguade, and Mateo Valero. Picos, A Hardware Task-Dependence Manager for Task-Based Dataflow Programming Models. In *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pages 878–880. IEEE, jul 2017.

[8] Ying hao Xu, Miquel Vidal, Beñat Arejita, Javier Díaz, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, and Filippo Mantovani. Implementation of the K-means algorithm on heterogeneous devices: a use case based on an industrial dataset. In *ParaFPGA: Parallel Computing with FPGAs*, 2017.

[9] Antonio Filgueras, Miquel Vidal, Marc Mateu, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, Dimitris Theodoropoulos, Dionisios N. Pnevmatikatos, Paolo Gai, Stefano Garzarella, David Oro, Javier Hernando, Nicola Bettin, Alberto Pomella, Marco Procaccini, and Roberto Giorgi. The AXIOM Project: IoT on Heterogeneous Embedded Platforms. *IEEE Design & Test*.

[10] Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, and Xavier Martorell. Exploiting Parallelism on GPUs and FPGAs with OmpSs. *1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems*, 2017.

[11] Jairo Balart, Alejandro Duran, Marc Gonzalez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Nanos Mercurium: A research compiler for OpenMP. *European Workshop on OpenMP (EWOMP'04)*, pages 103–109, 2004.

[12] Germán Llort, Antonio Filgueras, Daniel Jiménez-González, Harald Servat, Xavier Teruel, Estanislao Mercadal, Carlos Álvarez, Judit Giménez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. The Secrets of the Accelerators Unveiled: Tracing Heterogeneous Executions Through OMPT. In Naoya Maruyama, Bronis R. de Supinski, and Mohamed Wahib, editors, *OpenMP: Memory, Devices, and Tasks: 12th International Workshop on OpenMP, IWOMP 2016, Nara, Japan, October 5-7, 2016, Proceedings*, pages 217–236. Springer International Publishing, Cham, 2016.

[13] Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P. Singh. From OpenCL to high-performance hardware on FPGAs. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 531–534. IEEE, aug 2012.

[14] Intel FPGA SDK for OpenCL Programming Guide, 2017.

[15] SDAccel Development Environment Help, 2017.

[16] OpenMP Application Programming Interface - OpenMP Standard 4.5, 2015.

[17] Daniel Cabrera, Xavier Martorell, Georgi Gaydadjiev, Eduard Ayguadé, and Daniel Jiménez-González. OpenMP extensions for FPGA accelerators. In *2009 International Symposium on Systems, Architectures, Modeling, and Simulation*, pages 17–24. IEEE, jul 2009.

[18] Lukas Sommer, Jens Korinth, and Andreas Koch. OpenMP device offloading to FPGA accelerators. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 201–205. IEEE, jul 2017.

[19] Seyong Lee, Jungwon Kim, and Jeffrey S. Vetter. OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 544–554. IEEE, may 2016.